Eine objekt-orientierte Datenverwaltung für eine verteilte Visualisierungsumgebung

Von der Fakultät für Energietechnik der Universität Stuttgart
zur Erlangung der Würde eines
Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

vorgelegt von
Andreas Wierse
geboren in Trier

Hauptberichter : Prof. Dr.-Ing. Roland Rühle
Mitberichter : Prof. Dr.-Ing. Eberhard Göde
Tag der Einreichung : 11. Oktober 2000
Tag der mündlichen Prüfung : 26. Oktober 2001

Anwendungen der Informatik im Maschinenbau (AIM),
Fakultät für Energietechnik
der Universität Stuttgart
2001

Meiner Frau und meinem Sohn

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Rechenzentrum der Universität Stuttgart (RUS) und am Institut für Computeranwendungen II (ICA II). Mein besonderer Dank gilt Herrn Professor Dr.-Ing. Roland Rühle, der das RUS und die Abteilung Computersimulation und Visualisierung des ICA II leitet, für seine Unterstützung sowie für die Übernahme des Hauptberichtes und die Betreuung des Promotionsverfahrens an der Universität Stuttgart.

Mein Dank gilt ebenfalls Herrn Professor Dr.-Ing. Eberhard Göde vom Institut für Strömungsmechanik und Hydraulische Strömungsmaschinen der Universität Stuttgart für die freundliche Übernahme des Mitberichtes und seine Unterstützung.

Diese Arbeit entstand als Teil des Software-Systems COVISE und ist daher nur möglich gewesen durch die großartige Zusammenarbeit mit der Entwicklergruppe in der Abteilung Visualisierung des RUS. Besonders hervorzuheben ist Herr Dr. Ulrich Lang, der von Anfang an in ausführlichen Gesprächen und vielen Diskussionen als Sparringspartner wesentlich dazu beitrug, daß die Arbeit ihren heutigen Stand erreicht hat. Die Arbeit der Kern-Gruppe der COVISE-Entwickler (Ruth Lang, Harald Nebel, Daniela Rainer, Dirk Rantzau, Andreas Werner und Uwe Wössner) ermöglichte dann, die Ergebnisse dieser Arbeit in einer anwendungsnahen Weise verifizieren zu können.

Ferner gilt mein Dank allen Kolleginnen und Kollegen des RUS sowie den Mitarbeitern der Firma Cray Research, die mich insbesondere bei der Durchführung der Messungen und den sich daraus ergebenden Optimierungen unterstützt haben.

Schließlich bedanke ich mich bei meinen Eltern, ohne deren Unterstützung mein Ausbildungsweg und diese Arbeit nicht möglich gewesen wären.

Warmbronn, im Dezember 1999 Andreas Wierse

Zusammenfassung

Visualisierungssysteme die mit den großen Datenmengen arbeiten, die heute bei numerischen Simulationen anfallen, müssen mit diesen Daten effizient umgehen können. In dieser Arbeit werden die Anforderungen untersucht, die an die Datenverwaltung eines solchen Visualisierungssystems gestellt werden, das für die Darstellung der Ergebnisse komplexer numerischer Berechnungen nötig ist. Dazu werden diese Anforderungen anhand eines heute in der Industrie üblichen Falles skizziert und auf ihre Auswirkungen bei der Visualisierung hin beleuchtet.

Anschließend werden die Systemarchitektur des Visualisierungssystems COVISE sowie die es direkt beeinflussenden Systeme kurz vorgestellt. COVISE dient als Prototyp, innerhalb dessen die meisten der in dieser Arbeit vorgestellten Konzepte realisiert sind. Anschließend werden einige verbreitete Visualisierungssysteme anhand der anfangs festgestellten Anforderungen auf ihre Datenverwaltung hin analysiert und dem in COVISE realisierten Konzept gegenübergestellt.

Im darauffolgenden Kapitel werden diese Konzepte ausführlich vorgestellt. Die verteilte Datenverwaltung steht dabei im Mittelpunkt: es wird beschrieben, wie der Datenaustausch zwischen den Modulen realisiert ist, sowohl im lokalen Fall, als auch im Fall einer verteilten Anwendung. Die Vorteile dieses Ansatzes und die nötigen Mechanismen um einen eindeutigen Datenzugriff in einem verteilten System sicherstellen zu können werden dargelegt.

Den Datenobjekten kommt in diesem Zusammenhang eine Schlüsselrolle zu. Die für die Handhabung der Berechnungs- und Geometriedaten wichtigen Strukturen werden entwickelt und die passenden Hierarchien dargestellt. Im Hinblick auf die Nutzung dieser Datenobjekte durch die Module werden die Handhabung der Datenobjekte und die Zugriffsmechanismen beschrieben. Dabei werden auch die Möglichkeiten diskutiert, Datenobjekte persistent zu gestalten oder zu partitionieren.

Zum Abschluß dieses Kapitels werden einige praxisorientierte Betrachtungen angestellt: die Realisierung des Konzeptes im besonderen Umfeld einer High-Performance-Umgebung wird unter den beiden Aspekten Supercomputer und Netzwerke beschrieben. Die wesentlichen Strategien bei der Implementierung gehören ebenso dazu.

Die Anwendung des Datenverwaltungs-Konzeptes auf reale Probleme wird im vorletzten Kapitel untersucht. In drei realen Fällen wird die Praktikabilität des Ansatzes gezeigt: die verteilte Visualisierung der Berechnung einer Wirbelauflösung, der Strömung in einem Motorinnenraum und der Strömung im Saugrohr einer Turbine. Da für die Praxis die Performance eine nicht unerhebliche Bedeutung hat, folgt eine Reihe von Messungen die unter verschiedensten Bedingungen die Leistungsfähigkeit des Konzeptes demonstrieren.

Abschließend erfolgt eine Bewertung der Arbeit sowie ein Ausblick über die Möglichkeiten, die das Konzept über die im Rahmen dieser Arbeit realisierten Aspekte bietet.

Abstract

Visualisation systems that work on large amounts of data, as the ones produced by numerical simulations, must deal with this data efficiently. In this thesis the requirements will be examined that the data management of such a visualisation system must fulfill in order to perform the visualisation of the results of complex numerical simulations. These requirements are assessed for a standard application case of todays industry and will then be viewed under the aspect of their effect on the visualisation.

Next the system architecture of the COVISE visualisation system and the systems that influenced its development will be presented. COVISE serves as a prototype that is used to implement most of the concepts presented in this work. In addition several widely used visualisation systems will be analysed according to the initially defined requirements for the data management. This will be compared to the concept that has been realised in COVISE.

These concepts will be presented in detail in the following chapter. Here the focus is on the distributed data management: it is described how the data exchange between the modules is realised, for the local case as well as for a distributed scenario. The advantages of this approach will be explained as well as the necessary mechanisms to assure an unambiguous data access in a distributed system.

In this context the data objects play a key role. The structures that are important for the management of the computational and geometrical data are developed and the corresponding hierarchies are shown. With regard to the use of these data objects through the modules the handling and the access mechanisms are described. Here also possibilities are discussed to make data objects persistent or to partition them.

To finalise this chapter it is reflected on some practically oriented thoughts: the realisation of this concept in the special surrounding of a high-performance environment is described, taking especially the aspects supercomputing and networking into account. The main strategies for the implementation also belong here.

The application of the data management concept to real problems will be discussed in the subsequent chapter. The practicability of the approach will be shown in real applications: the distributed visualisation of a vortex breakdown, the flow in an internal combustion engine and the flow in a water turbine. Since performance is an important issue for the practical use, a number of measurements under different conditions will demonstrate the potential of the concept.

Finally an evaluation of the work will be given and an outlook will be offered on the possibilities that the concept opens up beyond the frame of the aspects realised in this thesis.

Inhaltsverzeichnis

Vorwort V
Zusammenfassung VII
Abstract VIII
Inhaltsverzeichnis IX
Abbildungsverzeichnis XIII
Tabellenverzeichnis XV

Kapitel 1
Einleitung 1

1.1 Die Problemstellung 1
1.1.1 Die Daten 1
1.1.2 Die Visualisierung 1
1.1.3 Die Verteilung 2
1.2 Zielsetzung der Arbeit 3
1.3 Gliederung der Arbeit 5

Kapitel 2
Die COVISE Visualisierungsumgebung 7

2.1 COVISE 7
2.2 Die Systemarchitektur 7
2.3 Motivation des Verteilungsmechanismus 9
2.4 RSYST 12
2.5 GRAPE 14

Kapitel 3
Vergleich mit existierender Software 17

3.1 Visualisierungssoftware 17
3.1.1 AVS 18
3.1.2 AVS/Express 20
3.1.3 IRIS-Explorer 21
3.1.4 Data-Explorer 22
3.1.5 Ensight 23
3.2 Übersichtstabelle 24

Kapitel 4
Konzept eines verteilten Visualisierungssystems 27

4.1 Die verteilte Datenverwaltung 28
4.1.1 Der Datenaustausch zwischen Modulen 28
4.1.2 Vorteile der fehlenden Direktverbindung 31
4.1.3 Mechanismen zur Konfliktvermeidung 31
4.2 Die Datenobjekte 34
4.2.1 Anforderungen an die Speicherung der Datenobjekte 34
4.2.2 Hierarchie der Datentypen 35
4.2.3 Zugriff eines Moduls auf die Datenobjekte 37
4.3 Persistenz 40
4.3.1 Definition von Persistenz 40
4.3.2 Auswirkungen der Persistenz 40
4.3.3 Vorteile der Persistenz 43
4.4 Partitionierte Objekte 43
4.4.1 Definition der Partitionierung 43
4.4.2 Gründe für die Partitionierung 44
4.4.3 Die Synchronisierung des Zugriffes auf Partitionen 46
4.4.4 Datentypen und Visualisierungsalgorithmen bei der Partitionierung 47
4.5 High-Performance-Komponenten 51
4.5.1 Supercomputer 51
4.5.2 Netzwerke 53

4.6 Wesentliche Strategien der Implementierung 54
4.6.1 Die Programmiersprache und verwendete Systemfunktionen 54
4.6.2 Objekt-Orientierung 55
4.6.3 Zugriff auf die Datenobjekte 57

Kapitel 5
Resultate 67

5.1 Anwendungsbeispiele 67
5.1.1 PEGASE: Berechnung einer Wirbelauflösung 67
5.1.2 Strömung im Zylinder eines Motors 68
5.1.3 Strömung in einem Saugrohr 72
5.2 Messungen 74
5.2.1 Messungen zwischen zwei Indigo Workstations 76
5.2.2 Messungen zwischen den beiden Cray Vektor-Supercomputern 79
5.2.3 Messungen zwischen einer Workstation und einer Cray 82
5.2.4 Fazit der Messungen 88

Kapitel 6
Bewertung und Ausblick 91

6.1 Bewertung 91
6.2 Ausblick 92

Kapitel 7
Literatur 95

Abbildungsverzeichnis

Abbildung 1-1: Die Visualisierungspipeline 2
Abbildung 2-1: Die COVISE System-Architektur 8
Abbildung 2-2: Bildschirmabzug der COVISE-Benutzungsschnittstelle 9
Abbildung 2-3: Eine Anfrage über den Object Request Broker (ORB) 11
Abbildung 2-4: Komponenten des integrierten Programmsystems RSYST 12
Abbildung 3-1: Die Kommunikation von AVS 18
Abbildung 3-2: Datentransfer bei AVS über Rechnergrenzen hinweg 19
Abbildung 3-3: Die Systemarchitektur des IRIS-Explorer 22
Abbildung 3-4: Kommunikation des IBM Data Explorer 23
Abbildung 4-1: Die COVISE Systemarchitektur 27
Abbildung 4-2: Lokaler Datenaustausch 28
Abbildung 4-3: Nicht-Lokaler Datenaustausch 29
Abbildung 4-4: "Deadlock"-Situation 32
Abbildung 4-5: Zugriff auf den Shared Data Space 37
Abbildung 4-6: Partitionierung eines 9 x 9 strukturierten Gitters 48
Abbildung 4-7: Uneindeutigkeiten bei Marching Cubes 50
Abbildung 4-8: Modul-Request Broker auf einem Supercomputer 53
Abbildung 4-9: Ausschnitt aus der Klassenhierarchie 56
Abbildung 4-10: Die Klassen des Request Brokers 56
Abbildung 4-11: Zugriff auf den Shared Data Space 58
Abbildung 4-12: Klassenhierarchie der Zugriffselemente 59
Abbildung 5-1: Der Datenfluß bei der Darstellung der PEGASE-Ergebnisse 68
Abbildung 5-2: Bildschirmabzug des Motordatensatzes in der COVISE-Umgebung 69
Abbildung 5-3: Leistungsfähigkeit von Rechnern und Netzwerk 69
Abbildung 5-4: Verteilung der Module und Daten 70
Abbildung 5-5: Datenobjekt-Anzeige der Partikelbahn 72
Abbildung 5-6: Strömung im Saugrohr einer Wasserturbine 73

Abbildung 5-7: Umgebung für die Messungen 74
Abbildung 5-8: Meßpunkte der Konfiguration Indigo/Indigo 76
Abbildung 5-9: Graphik der Meßergebnisse aus Tabelle 5-1 78
Abbildung 5-10: Meßpunkte der Konfiguration Y-MP/C94 79
Abbildung 5-11: Meßpunkte der Konfiguration Indigo/C94 82
Abbildung 5-12: Graphik der Meßergebnisse aus Tabelle 5-9 85
Abbildung 5-13: Graphik der Meßergebnisse aus Tabelle 5-10 87

Tabellenverzeichnis

Tabelle 3-1: Definition der Vergleichskriterien 24
Tabelle 3-2: Vergleich der Visualisierungssysteme 25
Tabelle 5-1: Indigo zu Indigo, FDDI 77
Tabelle 5-2: C94 zu Y-MP, FDDI 79
Tabelle 5-3: C94 zu Y-MP, HIPPI 80
Tabelle 5-4: C94 zu Y-MP, HIPPI, asynchrones write 81
Tabelle 5-5: C94 zu Y-MP, asynchrones write, optimale Socket-Parameter 81
Tabelle 5-6: C94 zu Indigo, Ethernet, kleiner Datensatz 83
Tabelle 5-7: C94 zu Indigo, FDDI, kleiner Datensatz 83
Tabelle 5-8: C94 zu Indigo, FDDI, großer Datensatz 84
Tabelle 5-9: C94 zu Indigo, FDDI, großer Datensatz, leere C94 85
Tabelle 5-10: Y-MP zu Indigo, HIPPI nach FDDI 86
Tabelle 5-11: Y-MP zu Indigo, HIPPI nach FDDI, leere Y-MP 87

Kapitel 1

Einleitung

1.1 Die Problemstellung

1.1.1 Die Daten

Die Ergebnisse von Berechnungen auf den schnellsten verfügbaren Supercomputern finden immer mehr Eingang in das tägliche Leben. Das reicht von Berechnungen zur Sicherheit der Fahrgastzelle von Automobilen bei Unfällen über die Qualität der Verbrennung in Ottomotoren, die extrem komplexen und immer zeitkritischen Wettervorhersagen bis zur Strömungsberechnung bei Flugzeugen zur Reduzierung des Treibstoffverbrauchs und der Schallemissionen. Da aber die zugrundeliegenden natürlichen Phänomene nur selten in einfache Formeln zu fassen sind und sich daher auch nur mit erheblichem Rechenaufwand numerisch simulieren lassen, besteht ein immer größerer Bedarf nach Rechenleistung. Siehe auch [18], [50] und [30].

Als Beispiel soll hier die numerische Simulation der Verbrennung in einem Ottomotor dienen. Die Wissenschaft ist heute weit davon entfernt, ein umfassendes Modell für die Vorgänge im Zylinder eines solchen Motors zu haben. Die heute verfügbaren Modelle beinhalten bereits starke Vereinfachungen. Dazu gehört zum Beispiel eine Beschreibung der chemischen Vorgänge, die sich auf die wesentlichen Elemente/Verbindungen und wiederum auf die wesentlichen Reaktionen zwischen diesen Elementen/ Verbindungen beschränkt. Das heißt, bevor man auch nur die Numerik betrachtet, hat schon eine drastische Vereinfachung des Modells stattgefunden. Hinzu kommen weitere Vereinfachungen hinsichtlich der Diskretisierung der Motorgeometrie und der Behandlung der Turbulenz, die in einer Motorinnenraumströmung auftreten. Trotz all dieser Vereinfachungen dauert die Berechnung eines vollen 4-Takt-Zyklus eines Otto-Motors auf schnellen Workstations etwa 30 Tage. Auf einem Supercomputer reduziert sich dies zwar deutlich, nimmt aber immer noch viele Stunden in Anspruch. Je nach Komplexität können sogar mehrere Tage vergehen.

Direkte Konsequenz dieser großen Rechenzeiten und der komplizierten Geometrien ist eine große Menge an Daten. Im Beispiel des Motors kommt man bei etwa 1.000.000 Gitterpunkten, einem Datensatz pro Grad Kurbelwinkel (720 bei insgesamt zwei Umdrehungen) und mindestens zehn Werten pro Gitterpunkt (Koordinaten, Geschwindigkeit, Temperatur, Druck, Konzentration der wichtigsten Elemente/Verbindungen) auf 57.600.000.000 Bytes, also 57,6 GigaByte an Daten.

1.1.2 Die Visualisierung

Die einzige Möglichkeit, solche Datenmengen sinnvoll auszuwerten, besteht in der Regel darin, sie graphisch darzustellen, zu visualisieren [7] [17] [41]. Dazu werden eine ganze Reihe von Visualisierungstechniken angewendet, die sich in einer sogenannten „Visualisierungspipeline“ darstellen lassen

(siehe Abbildung 1-1). In dieser Visualisierungpipeline sind die Schritte, die zu einer solchen Visualisierung gehören, zu sehen [31].

Abbildung 1-1: Die Visualisierungspipeline

- An erster Stelle steht in diesem Beispiel die Simulation. Man kann diesen Schritt auch als Datengenerator oder Datenquelle bezeichnen. Entsprechend könnte dort statt der Simulation auch ein Eingabeschritt stehen, der Daten einliest, die nach einer früheren Rechnung in einer Datei gespeichert wurden.
- Der nächste und in Anbetracht der immensen Datenmengen sehr wichtige Schritt ist das Filtern. Hier findet alles statt, was den eigentlich Datensatz reduziert: Ausdünnen in den räumlichen Komponenten, Ausdünnen in zeitlicher Richtung (Weglassen von Zeitschritten), Abschneiden der Geometrie, Extrahierung von Schnittflächen etc.
- Im Mapper werden die berechneten Daten in Geometriedaten umgerechnet oder abgebildet. Bei einer Schnittfläche wird z.B. der Temperaturwert als Farbe auf diese Fläche abgebildet. Bei einer Partikelverfolgung werden Partikel in die Strömung geworfen und deren Weg berechnet; als Ergebnis erhält man eine Partikelbahn in Form eines Polygonzuges. Eine sehr verbreitete Methode besteht darin, Isoflächen, also Flächen gleichen Wertes, auf den zugrundeliegenden Daten zu berechnen. Auf die daraus resultierenden Polygone lassen sich noch weitere Daten als Farbwerte abbilden. In gewissem Sinne befindet sich hier der Kern des Visualisierungsvorgangs.
- Der darauffolgende Renderingschritt berechnet aus den ihm vorliegenden Geometriedaten (Koordinaten, Farben, Normalen) dann das resultierende Pixelbild.
- Abschließend wird dieses Pixelbild vom Displayschritt auf dem Bildschirm ausgegeben. In vielen Fällen werden Rendering- und Displayschritt in einem Schritt zusammengefaßt.

1.1.3 Die Verteilung

Die Visualisierungspipeline liegt in der Regel nicht komplett auf dem Rechner, auf dem auch die Simulation läuft. Dafür gibt es eine Reihe von Gründen. Einer der wesentlichen ist, daß Rechenzeit auf dem Supercomputer, auf dem die Numerik in der Regel am schnellsten läuft, im Vergleich zu der lokalen Workstation, die der Wissenschaftler an seinem Arbeitsplatz hat, spürbar teurer ist. Viele Visualisierungsalgorithmen sind auch nicht an die speziellen Gegebenheiten der Supercomputer angepaßt, so daß sie auf einer Workstation kaum langsamer laufen als auf dem Supercomputer. Darüberhinaus kann es durchaus vorkommen, daß das Antwortzeitverhalten des Supercomputers für die interaktive Nutzung zu langsam ist (bedingt z.B. durch entsprechend hohe Auslastung durch große Numerikberechnungen). Hinzu kommt, daß die Graphikhardware, die in Workstations heute zu finden ist, bereits sehr leistungsfähig ist und das rein software-mäßige Rendern direkt auf dem Supercomputer nur in wenigen Fällen sinnvoll ist (z.B. bei Volume-Rendering).

Auf der anderen Seite ist es wenig sinnvoll und in der Regel unmöglich, die oben erwähnten 57,6 GB auf eine Workstation zu laden. Selbst wenn man vom verfügbaren Speicherplatz absieht, dauert alleine der Transfer über eine Ethernet-Verbindung (theoretisch 10 Mbit/s) etwa 13 Stunden. Sogar bei der Nut-

zung des zehnmal so schnellen FDDI braucht man noch deutlich über eine Stunde, um die Daten vom Supercomputer auf die Workstation zu laden.

Die logische Konsequenz in diesem Fall ist, den Filterschritt auf den Supercomputer vorzuziehen. Die Gesamtrechenzeit dürfte davon nicht wesentlich beeinflußt werden, da hierbei keine umfangreichen Berechnungen anfallen. Je nach Art der Visualisierung kann der Benutzer die Daten entsprechend reduzieren und muß nur noch die unbedingt nötigen Daten über das Netz transportieren und auf seiner Workstation speichern. Je nach Komplexität und Aufwand der Visualisierungsalgorithmen kann es selbstverständlich auch sinnvoll sein, weitere Schritte der Visualisierungspipeline auf den Supercomputer vorzuziehen [32] [35].

Als einfaches Beispiel mag hier die Berechnung einer Strömung auf einem massiv-parallelen Supercomputer dienen. Für die Simulation wird das Berechnungsgebiet mittels Gebietszerlegung in einzelne Bereiche gegliedert, die dann jeweils von einem Prozessor bearbeitet werden. Nach Abschluß der Berechnungen liegen die Ergebnisdaten auf dem entsprechenden Knoten vor. Eine Isofläche läßt sich nun am einfachsten berechnen, indem man den Algorithmus auf jedem Knoten und den dort schon vorliegenden Daten ausführt, anstatt die Daten erst zu reduzieren und dann auf die Workstation zu übertragen. In Abhängigkeit von dem für die Berechnung der Isofläche benutzten Wert reduziert sich die Menge an Daten, die über das Netz transportiert werden müssen, erheblich.

Speziell an diesem Beispiel wird jedoch auch ein grundlegendes Problem bei der optimalen Verteilung interaktiver Visualisierung deutlich. Die Menge an Daten, die nach der Berechnung der Isofläche über das Netz geschickt werden muß, hängt direkt vom interaktiv eingestellten Isowert ab. Das heißt, je nachdem für welchen Isowert der Benutzer sich entscheidet, können entweder wenige oder gar keine Daten anfallen oder aber (fallspezifisch) Datenmengen, die in der Größenordnung des ursprünglichen Datensatzes liegen. Es gibt also keine einfachen Regeln, um den für die Verteilung der Daten über das Netz optimalen Punkt in der Visualisierungspipeline zu bestimmen. Dies macht es auch schwierig bis unmöglich, die Verteilung sinnvoll zu automatisieren.

1.2 Zielsetzung der Arbeit

In dieser Arbeit wird ein Konzept präsentiert, das eine Lösungsmöglichkeit für die in Kapitel 1.1 aufgezeigten Probleme darstellt. Da der Kern der in diesem Kapitel beschriebenen Probleme mit der Handhabung der Daten verknüpft ist, erscheint es sinnvoll, die Gebiete Visualisierung und Datenverwaltung zu integrieren. Die Datenmanipulationsmöglichkeiten, die eine Datenverwaltung bietet, sollen mit den graphischen Darstellungsmöglichkeiten eines Visualisierungssystems zu einem Werkzeug verknüpft werden, das dem Nutzer die vollständige Kontrolle über die Verteilung der Daten auf den genutzten Rechnern ermöglicht.

Aufgezeigt wird eine Möglichkeit, diese beiden Welten so miteinander zu verbinden, daß die Vorteile beider Bereiche zusammenkommen. Der Schwerpunkt liegt in dieser Arbeit dabei zum größten Teil auf der Visualisierungsseite: Wie können Ansätze, die aus dem Gebiet der Datenverwaltung kommen, für eine effizientere Visualisierung genutzt werden? Wie läßt sich die Handhabung der Daten in einer Visualisierungsumgebung mit Hilfe von Datenmanagementtechniken verbessern? Ist der durch solche Techniken zusätzlich eingeführte Aufwand gerechtfertigt?

Aus den Anforderungen, die an die gewünschte Visualisierungsumgebung gestellt werden (insbesondere die effiziente Nutzung verteilter Umgebungen), ergibt sich schnell, daß ein modularer Ansatz sehr gut geeignet ist, die damit verbundenen Probleme zu lösen [37]. Das heißt, die Visualisierungsfunktiona-

lität wird in Module zerlegt, die entweder auf demselben oder aber auch auf verschiedenen Rechnern ausgeführt werden können. Die daraus folgenden Besonderheiten bei der Integration der Visualisierung mit einer entsprechenden Datenverwaltung erfordern entsprechende Interfacetechnologien. In einem baustein-artigen System müssen die Schnittstellen, über die die verschiedenen Module auf die Daten zugreifen, wohldefiniert und geeignet implementiert sein.

Der modulare Ansatz erlaubt eine sehr einfache und klare Strukturierung der eigentlichen Visualisierungsaufgabe (siehe auch Abbildung 1-1). Nach einer Zerlegung der Visualisierung in die einzelnen Schritte sind die entsprechenden Aufgaben jedem Modul klar zugewiesen. Der innere Ablauf der Module gliedert sich im allgemeinen in drei Teile:

- zunächst müssen alle Eingabeparameter und Eingabedaten in den Modul gebracht werden
- auf Basis dieser Parameter und Daten führt der Modul dann seine eigentliche Arbeit durch
- abschließend wird das Ergebnis dieser Arbeit wiederum in Form von Ausgabedaten bereitgestellt, die die Eingabedaten für die nachfolgenden Module darstellen.

Um die Handhabung dieser Module so einfach wie möglich zu machen, ist es sinnvoll, den Modulprogrammierer von der Komplexität der verteilten Umgebung abzuschirmen. Der soeben dargestellte innere Ablauf macht schon deutlich, daß dies ohne große Probleme möglich ist; die einzigen Kontakte zur Außenwelt des Moduls finden im ersten und im letzten Teil statt. Das heißt, der Kern des Moduls, in dem auch der wesentliche Teil seiner Funktionalität angesiedelt ist, ist völlig unabhängig von der Verteilung der Visualisierungspipeline. Durch die entsprechende Gestaltung dieser Zugriffe auf die Daten ist es möglich, die Module durchgängig so zu gestalten, daß überhaupt keine Kenntnis über die Verteilung der vorausgehenden oder nachfolgenden Module nötig ist.

Nachdem nun die Module keinerlei Information über die Lokalisierung der Daten benötigen, muß eine entsprechende Instanz eingeführt werden, die diese Information verwaltet und dafür sorgt, daß alle Module direkt auf die jeweils benötigten Daten zugreifen können. In einem verteilten System muß diese Instanz den Überblick über alle im System vorhandenen Datenobjekte haben. Darüberhinaus muß sie in der Lage sein, jede Anfrage eines Moduls nach einem entsprechenden Datenobjekt zu bearbeiten und dem Modul die gewünschten Daten bereitzustellen. In Anlehnung an ähnliche Mechanismen wird diese Instanz im folgenden „Request Broker“ genannt.

Die Arbeit dieses Request Brokers besteht also im wesentlichen aus zwei Teilen:

- aus Modulsicht muß das Datenobjekt zugreifbar sein, d.h. der Request Broker muß auf Anfrage das Datenobjekt zur Verfügung stellen und dem Modul mitteilen, wo es zu finden ist,
- aus Sicht des Request Brokers muß eine Datenverwaltungsfunktion vorhanden sein, die erlaubt, die Anfragen der Module zu bearbeiten.

Aus Modulsicht muß der Zugriff auf die Datenobjekte in einer Weise gestaltet werden, daß der Modulprogrammierer möglichst ohne Umwege auf die gewünschten Dateninhalte zugreifen kann. Man kann in diesem Zusammenhang von einem sogenannten Datenzugriffsinterface sprechen. Dieses Interface muß die folgende Funktionalität abdecken:

- der Zugriff auf das Datenobjekt muß hergestellt werden, unabhängig davon, wo das Datenobjekt ursprünglich angelegt wurde,
- auf die Teile des Datenobjektes muß zugegriffen werden können, möglichst in einer Weise, die den Modulprogrammierer bei seiner eigentlich Arbeit, der Implementierung des Visualisierungsalgorithmus, nicht behindert,

- es muß möglich sein, neue Datenobjekte anzulegen und diese mit Daten zu füllen (diese Datenobjekte müssen anschließend natürlich anderen Modulen zur Verfügung stehen).

Um in einem verteilten System eine schnelle Antwort auf Anfragen von Modulen zu gewährleisten, ist es sinnvoll, nicht einen zentralen Request Broker einzuführen (was beim Netzwerkzugriff zu deutlichen Performanceeinbußen führt), sondern ein dezentrales Netz von jeweils lokalen Request Brokern auf jedem beteiligten Rechner aufzubauen. Dies stellt sicher, daß keine unnötige Zeit bei der Kommunikation über Rechnergrenzen hinweg verloren geht. Die Datenverwaltungsfunktion muß zwischen diesen Request Brokern koordiniert werden.

Aus den obigen Bedingungen ergibt sich, daß der Datenverwaltung innerhalb des gesamten Systems eine wichtige Rolle zukommt. Sie garantiert eine einfache Handhabung der Module, einen sauber kontrollierbaren Datenfluß und ist mit ausschlaggebend für die Leistung des Gesamtsystemes. Ziel dieser Arbeit ist, basierend auf diesen Annahmen ein für die Visualisierungssoftware COVISE geeignetes Konzept für die Integration von Visualisierung und Datenverwaltung zu entwickeln und zu realisieren.

1.3 Gliederung der Arbeit

Nach einer kurzen Vorstellung der COVISE-Softwareumgebung und ihres Hintergrundes in Kapitel 2 wird ein Überblick über die gebräuchlichsten Visualisierungssysteme gegeben und deren Datenmanagement und Eignung für verteilte Visualisierung betrachtet; dazu gehören auch ein Blick auf die Datenbankszene und Mechanismen zur Verteilung von Daten und Objekten (Kapitel 3). In Kapitel 4 wird das Konzept in allen Einzelheiten vorgestellt, wobei auch auf die Probleme aus Kapitel 1.1 und die entsprechenden Lösungen eingegangen wird. Um zu zeigen, daß dieses Konzept auch in der Praxis funktioniert, werden in Kapitel 5 dann Anwendungsbeispiele vorgestellt. Zu der umfassenden Betrachtung dort gehören auch die Ergebnisse von Messungen, die unter realistischen Bedingungen auf Produktionsrechnern durchgeführt worden sind. In Kapitel 6 werden schließlich die Ergebnisse dieser Arbeit diskutiert und ein Ausblick auf die weiteren Entwicklungsmöglichkeiten gegeben.

Kapitel 2

Die COVISE Visualisierungs-umgebung

In diesem Kapitel wird ein kurzer Überblick über die Historie von COVISE und seine Systemarchitektur gegeben. Darüberhinaus werden zwei Software-Entwicklungen vorgestellt, die das in dieser Arbeit entwickelte Konzept wesentlich mitgeprägt haben: das Datenbanksystem RSYST und die objektorientierte Visualisierungssoftware GRAPE.

2.1 COVISE

Der Gegenstand dieser Arbeit (die Integration von Visualisierung und Datenverwaltung) wird als Kern in der Visualisierungsumgebung COVISE eingesetzt, bzw. ist zusammen mit ihr entwickelt worden. COVISE ist eine Abkürzung für **CO**llaborative **VI**sualization and **S**imulation **E**nvironment. COVISE wurde seit 1993 zunächst im Rahmen der EU-Projekte PAGEIN (RACE II, Nr. 2031) und ADONNIS (ESPRIT, Nr. 9309) entwickelt und wird heute im Rahmen zahlreicher Projekte weiter ausgebaut. Diese Visualisierungsumgebung bietet auf der einen Seite eine ideale Plattform, um die Ergebnisse dieser Arbeit zu überprüfen; auf der anderen Seite kommen aus dieser Konstellation zahlreiche neue Impulse, die die Weiterentwicklung entscheidend beeinflussen. Insbesondere diente COVISE bei den Messungen in Kapitel 5 als Anwendungsumgebung.

2.2 Die Systemarchitektur

COVISE ist eine Gemeinschaftsentwicklung mehrerer Mitarbeiter des Rechenzentrums der Universität Stuttgart. Dabei wird die Entwicklung in mehreren relativ unabhängigen Bereichen parallel betrieben:

- zentrale Ablaufsteuerung (Controller)
- Benutzungsschnittstelle (Userinterface)
- Graphikdarstellung in einer Fensterumgebung (OpenInventor-basierter Renderer)
- Graphikdarstellung in einer Virtual-Reality-Umgebung (Performer-basierter Renderer)
- Visualisierungsfunktionalität (Module)
- Datenverwaltung und Kommunikation.

Der Inhalt dieser Arbeit stammt fast ausnahmslos aus dem letzten Bereich und schließt dabei die Datenzugriffsmechanismen der Module mit ein. Dieser Teil ist klar von den übrigen Teilen von COVISE entkoppelt, da er im wesentlichen die untere Ebene darstellt, auf der die anderen Teile aufbauen.

Da COVISE in den weiteren Kapiteln vielfach referenziert wird und als Gesamtumgebung den Rahmen für diese Arbeit bietet, soll zunächst ein Überblick über die Struktur von COVISE gegeben werden. Die Darstellung der Systemarchitektur ist in Abbildung 2-1 zu sehen (man vergleiche hierzu die Darstellungen der Architekturen anderer Produkte (Kapitel 3)). Jedes Rechteck steht für einen Prozeß.

- Zentrale Komponente im Sinne der Steuerung der Abläufe ist der *Controller*. Dieser hat zu jedem anderen Prozeß eine Socketverbindung um z.B. den Start eines Moduls zu veranlassen.
- UI bezeichnet die *Benutzungsschnittstelle* des Systems. Über diesen Prozeß steuert der Benutzer die Anwendung und bekommt ein Feedback über den momentanen Stand der Ausführung.
- Die Prozesse A, B, C, D und E stehen für *Anwendungsmodule.* Dies sind die Bausteine aus denen der Benutzer seine Anwendung zusammenbaut, ähnlich einem Flußdiagramm (siehe Abbildung 2-2 links). Diese Anwendungsmodule werden im weiteren einfach als Module bezeichnet.
- RB steht für *Request Broker*. Der Request Broker ist die zentrale Instanz für die Verwaltung der Daten. Der Wirkungsbereich eines Request Brokers ist im Gegensatz zu dem des Controllers auf den lokalen Rechner beschränkt (die Verbindung zum Controller ist für den Ablauf nur von untergeordneter Bedeutung und wird daher in den weiteren Abbildungen aus Gründen der Übersicht weggelassen).
- Ein Request Broker verwaltet jeweils einen *Shared Data Space* über den der Austausch von Daten zwischen den einzelnen Anwendungsmodulen abgewickelt wird. In der Regel gibt es auf jedem beteiligten Rechner einen Request Broker und einen Shared Data Space.

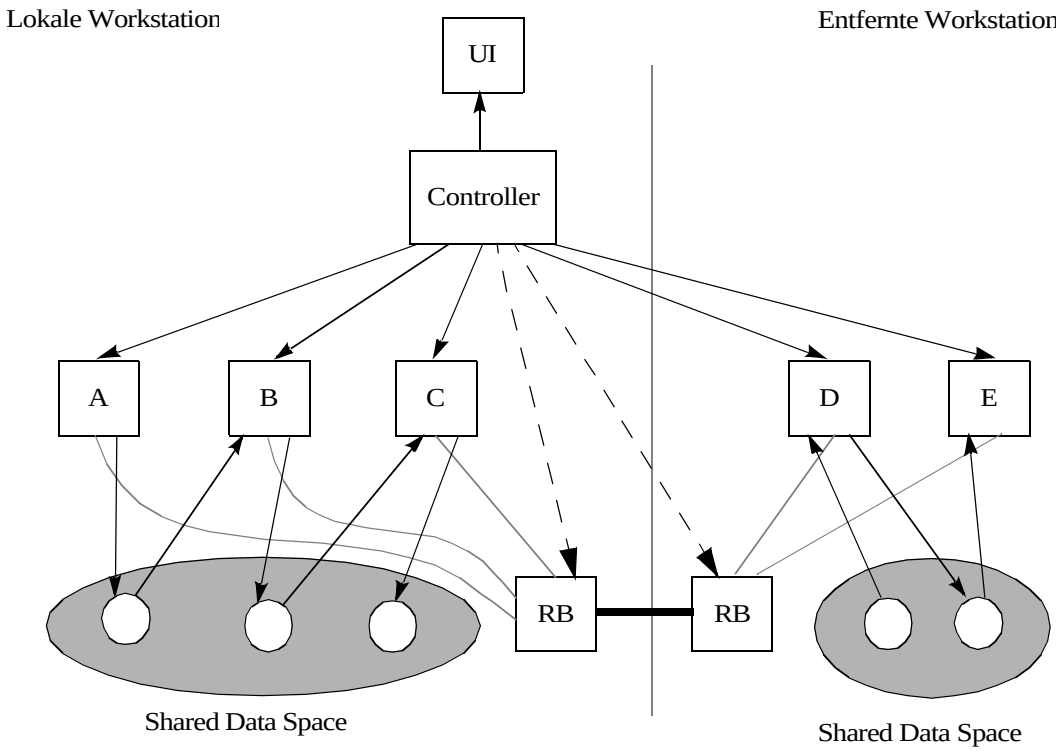

Abbildung 2-1: Die COVISE System-Architektur

Eine ausführliche Erläuterung der System-Architektur erfolgt in Kapitel 4, Konzept eines verteilten Visualisierungssystems.

Ein Bildschirmabzug einer Sitzung mit der COVISE-Software ist in Abbildung 2-2 zu sehen und zeigt die Benutzungsoberfläche und den Renderer. Deutlich zu erkennen ist im linken Fenster der einem Flußdiagramm ähnliche Aufbau der vom Benutzer zusammengesetzten Visualisierungsanwendung. In den vier Listen unmittelbar darüber sind die momentan zur Verfügung stehenden beteiligten Rechner aufgelistet, die Kategorien unter denen die Module einsortiert sind sowie die Module der gerade ausgewählten Kategorie. Die rechte Liste zeigt die Objekte, die beim aktuellen Ausführungszustand der Anwendung vorhanden sind. Durch Anklicken lassen sich diese Objekte detailliert inspizieren (siehe dazu auch Abbildung 5-5, Datenobjekt-Anzeige der Partikelbahn).

Im rechten unteren Fenster ist die OpenInventor-basierte Graphikdarstellung zu sehen. Auch in diesem Fenster gibt es eine Spalte in der die Objekte, auf denen diese Graphik basiert, zu sehen sind. Bei dem rechten oberen Fenster handelt es sich um ein Parameterfenster, in dem die Parameter einzelner Module gemeinsam dargestellt und vom Benutzer geändert werden können.

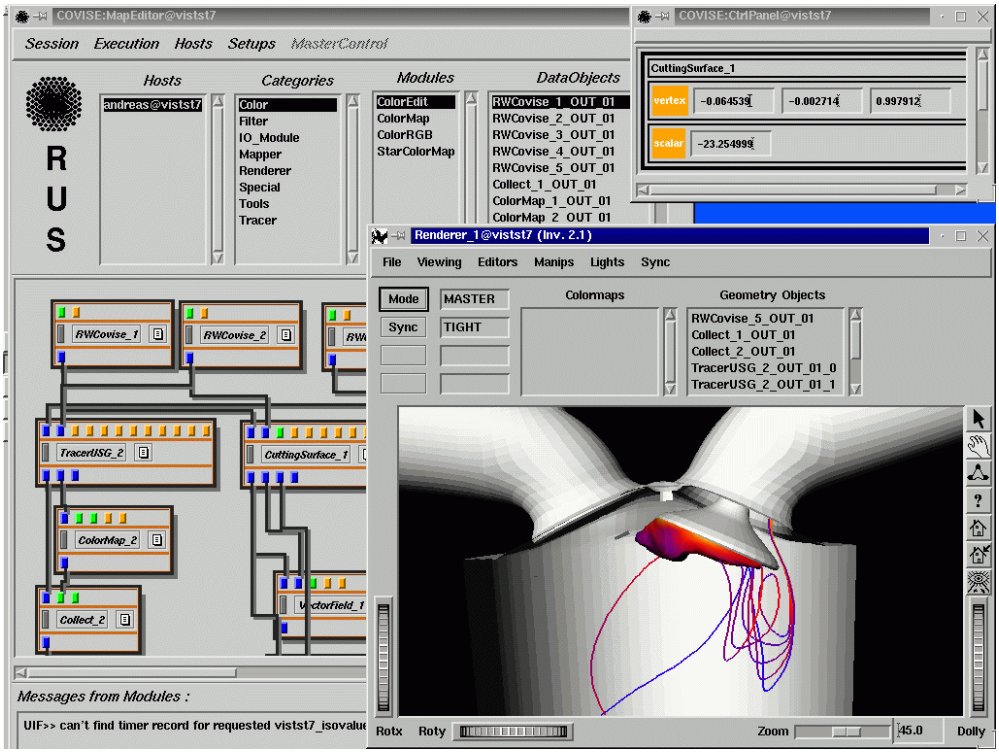

Abbildung 2-2: Bildschirmabzug der COVISE-Benutzungsschnittstelle

2.3 Motivation des Verteilungsmechanismus

Für die Verteilung von Objekten und den Zugriff auf diese sind heute zwei Verfahren weit verbreitet: DCOM von Microsoft und CORBA. Bei DCOM handelt es sich um ein proprietäres und bisher auf Unix-Rechner nicht verfügbares Verfahren. Da jedoch der Anwendungsbereich der Visualisierungsprobleme, die den Anstoß für diese Arbeit gaben, zum größten Teil auf eben diesen Unix-Rechnern liegt, wurde eine Lösung auf DCOM-Basis nicht weiter verfolgt. CORBA ist inzwischen auf einer großen Menge von Unix-Workstations verfügbar. Daher wird im folgenden ein Überblick über die Arbeitsweise von CORBA gegeben und dabei auch ein Vergleich mit den COVISE-spezifischen Anforderungen angestellt.

CORBA

CORBA ist eine Abkürzung für „Common Object Request Broker Architecture“, was sich übersetzen läßt mit „allgemeine Architektur zur Vermittlung von Objektanfragen“. Das zugrunde liegende abstrakte Objektmodel wie auch diese Architektur wurden von der Object Management Group (OMG) definiert. Zu dieser Gruppe gehören viele Hard- und Softwarehersteller, unter anderem Hewlett Packard, Compaq/ Digital Equipment, NCR und SunSoft. Dieses Objektmodell liefert eine strukturierte Darstellung von Objekt-Konzepten und Terminologie [69]. Entwickelt wurde CORBA, um die Komplexität neuer Softwareanwendungen zu reduzieren, die Kosten zu verringern und ihre Einführung zu beschleunigen. Dazu wurde eine Rahmenarchitektur mit detaillierten unterstützenden Schnittstellenspezifikationen geschaffen [46].

Der in [46] verwendete Begriff Objektsystem wird folgendermaßen definiert: eine Ansammlung von Objekten, die die Nachfrager von Services (Klienten/Clients) von den Anbietern der Services durch eine wohl-definierte kapselnde Schnittstelle isolieren. Genauer gesagt, werden Clients von der Implementierung von Services im Hinblick auf Datenrepräsentation und ausführbaren Code isoliert. Die Idee, die hinter diesem Konzept steht, besteht in einer völligen Trennung von der anfordernden Einheit und der ausführenden Einheit. Die Schnittstelle zwischen diesen beiden Einheiten wird detailliert definiert, so daß eine saubere Trennung sichergestellt ist.

Zunächst die Definition einiger Begriffe in CORBA:

- Objekte
  ein Objekt ist eine identifizierbare, gekapselte Einheit, die einen oder mehrere Dienste zur Verfügung stellt, die von einem Client angefordert werden.
- Anfragen
  Clients fordern einen Dienst mittels einer Anfrage an. Eine Anfrage ist ein Ereignis, d.h. etwas was zu einem gewissen Zeitpunkt passiert. Die Information, die zu so einer Anfrage gehört, besteht aus einer Operation (oder Methode), einem Teilobjekt, null oder mehr Parametern und optional einem Anfrage-Kontext.
- Objektimplementierung
  die Implementierung eines Objektes führt die Berechnungsaktivitäten durch, die nötig sind, um den von der Anfrage beabsichtigten Effekt herbeizuführen (z.B. berechnen des Ergebnisses und aktualisieren des Systemstatus).

Aus diesen Definitionen wird bereits der Unterschied zu dem in dieser Arbeit behandelten Daten- und Objektmodell deutlich. Während die CORBA-Architektur das Objekt in klassischer Manier in den Mittelpunkt aller Abläufe stellt und Anfragen in der Form von Methoden, die an ein Objekt geschickt werden, gehandhabt werden, werden in COVISE Objekte und Methoden (bzw. Module) gleichberechtigt nebeneinander gesehen.

Der Object Request Broker

Abbildung 2-3 zeigt, wie eine Anfrage von einem Client zur Implementierung des Objektes geschickt wird. Der Client ist die Einheit, die eine Operation auf einem Objekt durchführen möchte und die Objektimplementierung sind der Code und die Daten die tatsächlich das Objekt implementieren. Der ORB ist verantwortlich für alle Mechanismen, die nötig sind, die passende Objektimplementierung zur Anfrage zu finden, die Implementierung für den Empfang der Anfrage vorzubereiten und die Daten, die diese Anfrage ausmachen, zu kommunizieren.

Das Interface, das der Client sieht, ist völlig unabhängig davon, wo das Objekt liegt, welche Programmiersprache benutzt wurde, es zu implementieren oder von jedem anderen Aspekt, der nicht die Schnittstelle des Objektes betrifft. Der Object Request Broker zeichnet sich also dadurch aus, daß er einem Client ermöglicht, Methoden an Objekte zu schicken, von denen er nicht weiß (und auch nicht wissen möchte) wo sie liegen. Ein Client kennt nur die logische Struktur eines Objektes basierend auf der Schnittstelle und ihren Methoden und erfährt das Verhalten des Objektes durch den Aufruf dieser Methoden. Dabei kann ein Client durchaus auch ein Objekt sein, das von anderen Clients mit Anfragen zu Methodenaufrufen veranlaßt werden kann.

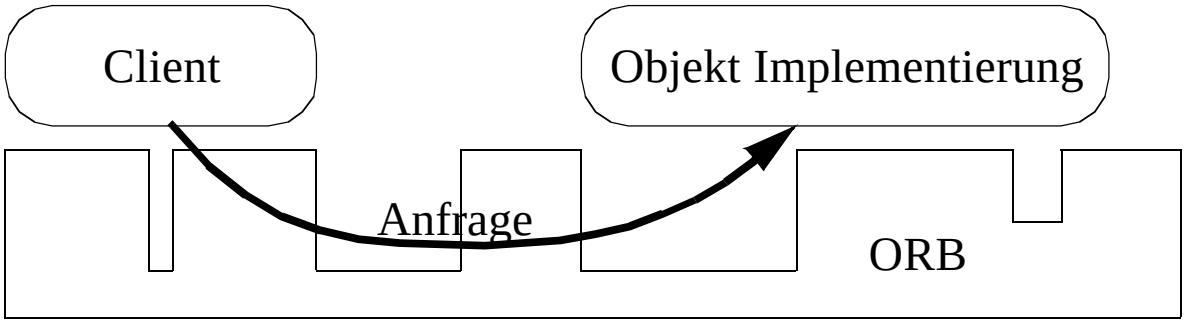

Abbildung 2-3: Eine Anfrage über den Object Request Broker (ORB)

CORBA und COVISE

Bei CORBA handelt es sich um ein sehr sauberes Modell für verteilte Objekte mit großem Gewicht auf Transparenz und Flexibilität. Dieser sehr generelle Ansatz führte dazu, daß lange Zeit keine Implementierungen von CORBA ORBs verfügbar waren und auch die heute verfügbaren nur auf wenigen Plattformen laufen. Hinzu kommt, daß durch die sehr generelle Auslegung ein signifikanter Overhead bedingt ist, der auch bei den verfügbaren ORBs deutlich wird: CORBA ist langsam. Die zweite Generation der ORBs geht zwar dieses Problem an und ist auch deutlich schneller als die erste Generation, liegt aber immer noch deutlich unter dem Optimum. Die Umgebung, in der COVISE zum Einsatz kommt, zeichnet sich dagegen durch große Datenmengen aus, die in zum Teil sehr speziellen Rechnerumgebungen möglichst effizient gehandhabt werden müssen. Allein durch diese Randbedingungen ist CORBA nicht die geeignete Architektur zur Handhabung der verteilten Objekte.

Dies sind jedoch nicht die einzigen Gründe, die zur Entwicklung eines eigenen Datenmanagements geführt haben. Vielmehr sind die Anforderungen, die an ein Datenmanagement in einem Visualisierungswerkzeug für die Darstellung von Simulationsergebnissen gestellt werden, deutlich anders als die, die man an eine „generelle Architektur zur Vermittlung von Objektanfragen“ stellt. Von wesentlicher Bedeutung ist hier insbesondere, daß man gerade wenn man mit sehr großen Mengen von Daten arbeitet, sehr genau wissen muß, wo sich diese Daten befinden, beziehungsweise wo neue Daten erzeugt werden sollen. So kann es zum Beispiel durchaus sinnvoll sein, eine Isoflächenberechnung für ein besonders komplexes Gebiet auf einem Supercomputer durchzuführen, während sie im Normalfall (vielleicht aus Kostengründen) auf der Workstation des Benutzers durchgeführt wird.

Es muß also eine Möglichkeit geben, eine Berechnung einem bestimmten Computer zuzuweisen. Dadurch ergibt sich abweichend vom klassischen Objekt/Methoden Modell, bei dem das Objekt den Ort bestimmt, an dem die Methode ausgeführt werden soll, ein Methoden-orientiertes Modell, bei dem die Methode auf einem bestimmten Rechner ausgeführt werden soll und die Daten dann zur nächsten

Methode geschickt werden (die durchaus auch auf einem anderen Rechner „liegen“ kann). Besonders deutlich wird dies, wenn man sieht, wie Visualisierungsvorgänge im allgemeinen beschrieben werden (siehe auch Abbildung 1-1, Die Visualisierungspipeline): es wird eine Prozedur beschrieben, mit der man von einem Ausgangsdatenobjekt zu einem Bild gelangt. Dabei stehen die Methoden im Mittelpunkt und nicht die als Zwischenrepräsentation entstehenden Objekte.

Ein weiterer Aspekt ist die Performance eines solchen verteilten Systems. Damit ist noch nicht einmal die generelle Performance von CORBA Implementierungen gemeint, sondern vielmehr die speziell an die High-Performance Umgebung anzupassenden Parameter z.B. von Netzwerkverbindungen. Messungen (siehe Abschnitt 5.2) haben ergeben, daß bei HIPPI-Verbindungen die Standardparameter den maximalen Durchsatz auf knapp über 10% des theoretisch möglichen Durchsatzes beschränkten. Bei der Optimierung der Verbindungsparameter konnten bis zu 70% des theoretisch möglichen Durchsatzes erreicht werden. Das heißt, selbst wenn für die benutzten Rechner CORBA-Implementierungen verfügbar gewesen wären, wäre die Effizienz, mit der diese Implementierungen die Infrastruktur ausnutzen, unbefriedigend gering gewesen.

2.4 RSYST

Am Institut für Kernenergetik und Energiesysteme und später am Rechenzentrum der Universität Stuttgart wird seit 1969 das Softwaresystem RSYST entwickelt, das als offenes Daten-, Methoden- und Modellbanksystem die Integration von Programmen und Daten in Ingenieurs- und wissenschaftlichen Berechnungen zum Ziel hat [24] [36] [53] [73]. Bei RSYST handelt es sich nicht um eine Datenbank im herkömmlichen Sinne. Vielmehr ist die Datenbasis in RSYST die zentrale Komponente, um die herum die übrigen Teile angelegt sind: der Benutzer baut eine Folge von Modulen auf, die ihre Daten über eine Datenbank austauschen. Die Daten stehen also ganz zentral und die einzelnen Funktionen, die der Benutzer ausführen möchte, werden auf sie angewandt. RSYST ist im Kern ein monolithisches nicht-verteiltes System; die verschiedenen Komponenten in Abbildung 2-4 entsprechen daher nicht verschiedenen Prozessen.

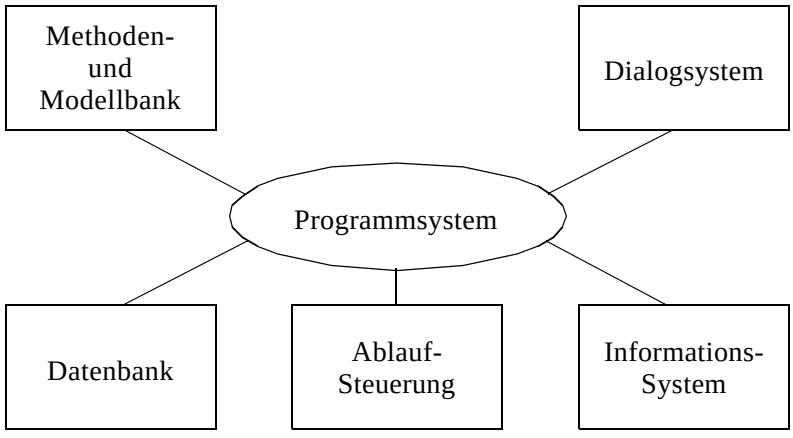

Abbildung 2-4: Komponenten des integrierten Programmsystems RSYST

Die Ansätze, RSYST auf verteilte Umgebungen anzupassen, basierten auf dem sogenannten Remote Procedure Call, das heißt Funktionen auf anderen Rechnern werden über eine bestimmte Schnittstelle ähnlich Programm-internen Funktionen aufgerufen [14] [63]. Die Daten, die mit diesen Funktionen ausgetauscht werden, werden in Form von Argumenten übergeben. Die Datenbasis bleibt dabei lokal, es ist

kein direkter Zugriff vom Rechner, auf dem die Funktion ausgeführt wird, auf die Daten in der Datenbasis möglich.

Im Laufe der Entwicklung von RSYST wurden verschiedene Varianten zur Integration von Graphik- oder Visualisierungsfunktionalität entwickelt und implementiert. In [57] wird ein objekt-orientierter Ansatz vorgestellt, der graphische Primitive wie Kreise, Rechtecke, zwei- und dreidimensionale Diagramme als Objekte definiert. Diese Datenobjekte wurden in der RSYST-Datenbasis gehalten, so daß von entsprechenden Modulen auf diese Datenobjekte zugegriffen werden konnte. Dieses Vorgehen unterscheidet sich deutlich von der Objekt-Orientierung wie sie in COVISE aufgefaßt wurde. Die Objektgranularität liegt bei COVISE auf dem Niveau des Anwendungsproblemes: Datenobjekte sind hier ganze Berechnungsgitter, Geometrien oder Datenfelder. Die effiziente Handhabung dieser Datenobjekte als einzelne und transportable Objekte in einer verteilten Umgebung macht es erforderlich, die Granularität so grob wie möglich anzusetzen. Es ist nicht sinnvoll, jedes Graphikprimitiv einzeln über das Netzwerk zu schicken, wenn es auf einem anderen Rechner dargestellt werden soll.

Eine später (ca. 1991) entwickelte Visualisierungskomponente von RSYST wurde über einen Modul realisiert. Im folgenden wird der PHIGS-basierte Graphikmodul beschrieben; der Graphikstandard PHIGS ist zwar nahezu komplett von der Bildfläche verschwunden, aber die Kopplung von RSYST und PHIGS auch im Hinblick auf die Datenstrukturen ist im Hinblick auf das Thema dieser Arbeit sehr interessant. Die Visualisierung unter RSYST basiert wesentlich auf sogenannten graphischen Objekten [34]. Im Fenster des PHIGS-Moduls können ein oder mehrere solcher graphischen Objekte dargestellt werden. Ein graphisches Objekt ist dabei die Repräsentation einer gewissen Menge von Datenobjekten aus der RSYST Datenbasis. Anzahl und Typ dieser Datenobjekte ist fest für ein bestimmtes graphisches Objekt, Größe und Inhalt sind variabel. Nachdem der Typ des graphischen Objektes festgelegt worden ist, wird eine neue Instanz dieses Objektes basierend auf den Datenobjekt-Informationen, die aus der RSYST Datenbasis gelesen werden, erstellt. Ein solches graphisches Objekt kann dann auf verschiedene Weise manipuliert werden: entweder über Objekttyp-unabhängige Aktionen wie Drehen, Skalieren, Verschieben, etc. oder über die Änderung objekttyp-abhängiger Parameter wie zum Beispiel der Isowert für eine Isoflächenberechnung.

Sowohl die PHIGS Datenstruktur als auch die von RSYST sind hierarchisch organisiert. Daher bot es sich an, die PHIGS Datenstruktur auf die RSYST Datenbasis abzubilden; d.h. das Ergebnis der Visualisierung wird in der Datenbasis gespeichert. Dadurch ist es möglich, die Strukturen und Daten aus PHIGS mittels RSYST-Modulen zu manipulieren und das Ergebnis wiederum in der PHIGS-RSYST Umgebung darzustellen. Dabei ist es zum Beispiel möglich, nur das Muster einer Visualisierung (zum Beispiel Rotations- oder Translationsfaktoren, die Position von Lichtquellen oder Objekttyp-abhängige Parameter wie den Isowert) abzuspeichern.

In [62] wird ein Konzept zur Visualisierung von Mehrkörpersystemen mit RSYST als Integrationssystem vorgestellt. Hierbei werden verschiedene Komponenten gekoppelt, die für den Datenaustausch die RSYST-Datenbasis verwenden. Alle Module greifen auf die Datenobjekte, die zwischen ihnen ausgetauscht werden, über diese Datenbasis zu. Aus Geschwindigkeitsgründen werden bei den Optimierungen die Datenobjekte sowie die Strukturinformationen im Hauptspeicher gehalten. Um den Simulations- und Optimierungsmodul, der hier zur Anwendung kommt, auch auf andere leistungsfähige Rechner auslagern zu können, wird eine sogenannte Multiple-Access Datenbank eingesetzt, d.h. mehrere verteilte Berechnungsprogramme können gleichzeitig auf dieselbe Datenbank zugreifen.

Für diese Verteilung von Modulen wird der Modul DFNMDL eingesetzt. Dieser besteht aus drei Teilen: der Initialisierung und dem Start beliebig vieler Module, der Verbindung mit den im Wartestatus

befindlichen Modulen und dem Abbrechen der Verbindungen mit dem Beenden der Module. Der DFN-MDL-Modul basiert auf dem DFNRPC, eine vom DFN unterstützte Weiterentwicklung eines RPC [48]. Ein RPC-Werkzeug verteilt Anwendungen über prozedurale Schnittstellen, d.h. eine Funktion kann auf einen anderen Rechner ausgelagert werden. Der RPC stellt dazu die nötigen Konvertierungsroutinen und die Routinen für die Datenübertragung bereit. Bei jedem Aufruf wird dabei erneut die komplette Abfolge von Verbindungsaufbau, Datentransfer, Prozedur, Datenrücktransfer und Verbindungsabbau durchlaufen.

Für die Visualisierung der Mehrkörpersysteme werden zwei Visualisierungsmodule verwendet: für die Erstellung der Geometrie GEOEDIT, der aus parametrisierten Formen die Punkt-, Verbindungs- und Normalenvektoren berechnet, die für die Darstellung und Beleuchtung nötig sind. Für die leistungsfähige online und offline Visualisierung steht MBSVIS zur Verfügung. Dieser Modul ist auch in der Lage, Kraftelemente darzustellen, die während der Visualisierung ihre Geometrie verändern (wie z.B. Federn oder Dämpfer). Eine genaue Beschreibung von MBSVIS findet sich in [25].

RSYST und COVISE

Der wesentliche Unterschied zwischen RSYST und COVISE besteht darin, daß RSYST als Softwaresystem für wissenschaftliche Anwendungen entwickelt worden ist, dem die Visualisierung später als Modul hinzugefügt wurde. COVISE ist dagegen als verteiltes Visualisierungssystem konzipiert worden, dem eine gewisse Funktionalität für den Umgang mit den dabei verwendeten Daten mitgegeben wurde (hierbei sind viele Erfahrungen, die im Laufe der RSYST-Entwicklung gemacht wurden, eingeflossen [37]). Dementsprechend liegen auch die Stärken von RSYST bei der Datenbasis, die von COVISE in der Visualisierungsorientierung und der Verteilung.

2.5 GRAPE

Das objekt-orientierte Graphiksystem GRAPE (GRAphics Programming Environment) wird seit 1985 an der Universität Bonn sowie seit 1992 auch an der Universität Freiburg entwickelt (siehe [55], [17] und [74]). Basierend auf der Programmiersprache C wurde ein objekt-orientierter Kernel entwikkelt. Objekte in diesem System sind ähnlich wie in COVISE die Datenelemente mit denen der Benutzer arbeitet, also Finite Elemente Gitter, Geometrien oder ähnliches. Die Visualisierungsalgorithmen werden als Methoden direkt auf diese Datenobjekte angewendet. Dabei sind die verschiedenen Datentypen als Klassen definiert, die sich innerhalb einer Vererbungshierarchie befinden. Diese ist relativ flach, unterscheidet aber zum Beispiel zwischen Gittern (Triang3D) und Gittern mit Daten (Fe3D). Bei Geometrien gibt es eine Hierarchie von Punkten über Linien zu Polygonen und Volumina.

Da es sich bei GRAPE um ein monolithisches und nicht verteiltes System handelt, ist es relativ einfach, alle Visualisierungsalgorithmen direkt als Methoden auf den entsprechenden Datenobjekten zu implementieren. So gibt es zum Beispiel für die Klasse Fe3D eine Methode *cutting-plane*, die die Schnittfläche durch das Gitter berechnet und als Objekt der Klasse Surface zurückgibt. Durch Schicken der Methode *draw* wird anschließend das Ergebnis graphisch dargestellt. Die Visualisierungspipeline aus Kapitel 1 wird also in verschiedenen Schritten abgearbeitet, zwischen denen Datenobjekte erzeugt werden.

Diese Datenobjekte sind jedoch für den interaktiven Benutzer nicht zugreifbar. Vielmehr definiert dieser Paare von Datenobjekten und Methoden, die dann automatisch ausgeführt werden, wenn sich ein Parameter oder die Graphikdarstellung ändert. Die sehr enge Integration aller Algorithmen in Form von Methoden in eine ausführbare Datei erlaubt dabei eine sehr hohe Geschwindigkeit. Die Methoden befin-

den sich dabei in einem sortierten Baum und werden über einen ASCII-Namen aufgerufen. Die Datenobjekte liegen direkt in Form eines Zeigers vor.

Diese Strukturen machen GRAPE zu einem extrem effizienten Visualisierungswerkzeug, daß Parameteränderungen genauso schnell darstellt wie einfache geometrische Transformationen. Die konsequent durchgehaltene Objekt-Orientierung macht es sehr einfach, GRAPE zu erweitern und anzupassen. So lassen sich beispielsweise neue Graphikbibliotheken nach Implementierung weniger Methoden (z.B. *update*, *begin_patch*, *patch_normal*) schnell nutzen. Die Ausgabe in eine PostScript-Datei erfolgt beispielsweise über dieselbe *draw*-Methode wie in ein Graphikfenster; vorher wird nur das Graphik-Device auf PostScript gesetzt.

GRAPE und COVISE

Die Klassen für die einzelnen Datenobjekte sind in GRAPE und COVISE sehr ähnlich. Das zugrundeliegende Verständnis vom Verhältnis von Datenobjekten zu Methoden/Modulen ist im Prinzip dasselbe. GRAPE ist jedoch im Gegensatz zu COVISE ausschließlich für eine monolithische nicht-verteilte Nutzung konzipiert. Es findet sich keine Möglichkeit, Methoden auf anderen Rechnern auszuführen. Auch sind keine Vorkehrungen getroffen, die einzelnen Datenobjekte (die immerhin existieren) dem Benutzer in irgendeiner Form zugänglich zu machen.

Kapitel 3

Vergleich mit existierender Software

Im ersten Teil dieses Kapitels (3.1) soll ein Überblick über die erfolgreichsten auf dem Markt befindlichen Visualisierungsprogramme gegeben werden. Nach einer Vorstellung der prinzipiellen Funktionsweise und einem Überblick über die Architektur des jeweiligen Programmes wird kurz untersucht, wie den in Kapitel 1.1 geschilderten Problemen begegnet wird. Da die Menge der verfügbaren Programme zwar recht groß ist, in der Praxis aber nur sehr wenige weit verbreitet sind, konzentrieren wir uns hier auf die erfolgreichsten Systeme.

In Kapitel 3.2 findet sich eine Vergleichstabelle in der die einzelnen bis dahin angesprochenen Visualisierungssysteme einander gegenübergestellt werden.

3.1 Visualisierungssoftware

Im Bereich der Visualisierung hat sich in den letzten Jahren sehr deutlich das Prinzip des visuellen Programmierens durchgesetzt (visual programming paradigm) [9] [59]. Die Visualisierungsaufgaben werden dabei, analog der Darstellung in Abbildung 1-1, auf einzelne Bausteine verteilt. Hierzu wählt der Benutzer für jeden Verarbeitungsschritt (Filter, Mapper, Renderer, Display) aus einer Vielzahl von verschiedenen Algorithmen den für seine Visualisierung nötigen aus. Es genügt jedoch nicht, die zu verwendenden Algorithmen auszuwählen; vielmehr muß der Benutzer dem System auch noch genau mitteilen, in welcher Reihenfolge die Module, die die Algorithmen ausführen, abgearbeitet werden sollen und wie die Ergebnisse eines Moduls zum nächsten Modul gelangen.

Dazu sind die Module mit sogenannten Eingabe- und Ausgabeports versehen. An den Eingabeports werden die Daten angeliefert, die der Modul braucht um seine Arbeit auszuführen. Die Resultate seiner Arbeit stellt er dann an den Ausgabeports zur Weiterverarbeitung zur Verfügung. Durch Verbinden von Ausgabe- und Eingabeports bestimmt der Benutzer nun, welche Daten von welchem Modul verarbeitet werden.

Betrachtet man nun das gesamte Szenario, so baut der Benutzer aus den Modulen und durch das Verbinden der Ports ein Flußdiagramm auf, das seinen Visualisierungswunsch ausführt. In diesem Zusammenhang spricht man auch von einem Datenflußmodel (Dataflow Model). Viele Visualisierungspakete gehen sogar so weit, den Ablauf dieses Netzwerkes über diesen Datenfluß zu steuern, das heißt ein Modul beginnt dann mit der Ausführung, wenn alle benötigten Eingabeports mit Daten versorgt sind. Wenn der Modul dann fertig ist, stellt er seine Resultate an den Ausgabeports zur Verfügung, was wie-

derum die entsprechenden Eingabeports der nachfolgenden Module aktiviert. So wird implizit über die Bereitstellung von Resultaten der Ablauf gesteuert.

Die meisten der in diesem Kapitel besprochenen Programmpakete folgen diesem Ansatz. Die einzige Ausnahme stellt Ensight von CEI dar. Die Möglichkeit zur Verteilung von Funktionen ist bei Ensight nur sehr rudimentär; da Ensight jedoch ein weitverbreitetes Visualisierungssystem ist, wird es mit einer kurzen Betrachtung in diesen Vergleich mit einbezogen.

So einleuchtend und sinnvoll der Ansatz des reinen visuellen Programmierens auch erscheint, so bringt die Tatsache, daß Datenfluß und Ablaufsteuerung untrennbar miteinander verbunden sind, doch das ein oder andere Problem mit sich. Ein offensichtliches Problem ist, daß es keinen Weg gegen den Fluß gibt. Da die Ausführung eines Moduls über das Eintreffen eines neuen Datenobjektes an einem der Eingabeports ausgelöst wird, entsteht beispielsweise sofort eine Endlosschleife, wenn man den Ausgang eines Moduls mit dem Eingang eines vorher ausgeführten Moduls verbindet, da es keine natürliche Abbruchbedingung gibt. Aus diesem Grunde finden sich heute in den verbreiteten Programmen, die dem visuellen Programmieren folgen, fast immer Abwandlungen oder Erweiterungen des reinen Datenfluß-modelles.

3.1.1 AVS

AVS ist der Urahn des visuellen Programmierens in der wissenschaftlichen Visualisierung. Als erstes Programmpaket dieser Art, im wesentlichen entwickelt 1989 von Craig Upson und seiner Gruppe bei Stellar [71], ist es auch das ausgereifteste und verbreitetste. Inzwischen in der Version 5 verfügbar, hat es einen sehr hohen Entwicklungsstand erreicht und auch zahlreiche Verbesserungen erfahren. Die wesentliche Architektur hat sich seit Version 4 nicht mehr verändert [5] [6]. Parallel zu dieser Visualisierungssoftware gibt es die Neuentwicklung AVS/Express, die sich im Kern deutlich von AVS 5 unterscheidet. Die Architektur von AVS/Express wird in Kapitel 3.1.2 vorgestellt.

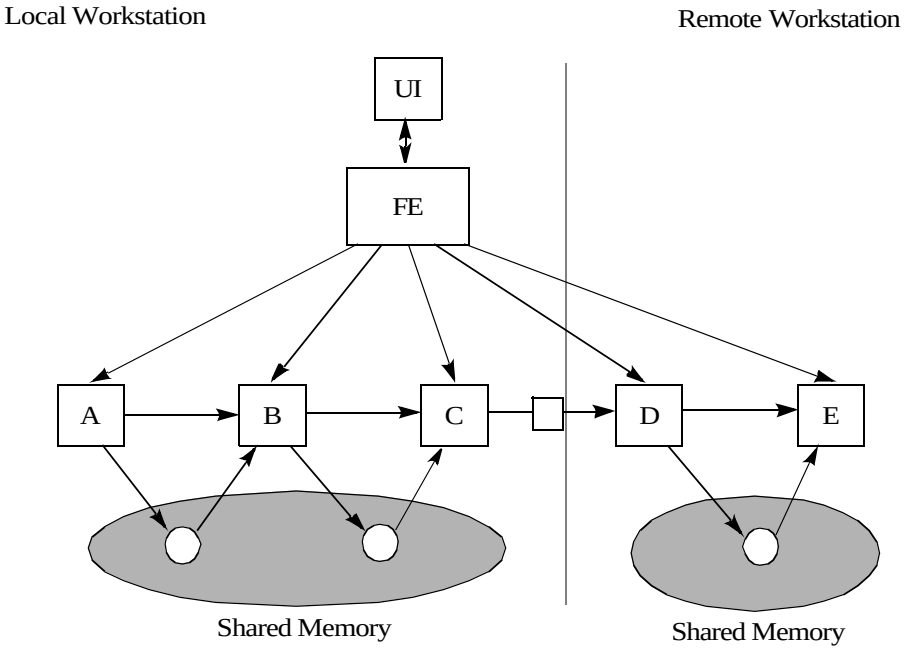

Abbildung 3-1: Die Kommunikation von AVS

In Abbildung 3-1 ist die Systemarchitektur von AVS in einer verteilten Umgebung dargestellt. Die zentrale Steuerung des Ablaufs wird von einem sogenannten *Flow Executive* (FE) übernommen. Dieser hat direkten Kontakt zu jedem Modul (A bis E) und zur Benutzungsschnittstelle (UI). Dort wird die Information über die Topologie des Netzwerkes gehalten, d.h. welche Modul Ausgabeports mit welchen Modul Eingabeports verbunden sind. Dies impliziert natürlich auch die Reihenfolge, in der die Module auszuführen sind. Der Flow Executive überwacht auch den Datenfluß, d.h. welche Daten wohin zu schicken sind und über welche Kanäle dies zu erfolgen hat. Die Parameter, die von einzelnen Modulen benötigt werden, werden vom Benutzer an der Benutzungsschnittstelle eingestellt und über den Flow Executive an die Module weitergereicht.

Bei AVS werden Module nicht notwendigerweise als einzelne Prozesse gestartet. Vielmehr sind häufig genutzte Module in einem gemeinsamen Executable untergebracht. Dies bringt auf der einen Seite den Vorteil, daß der Rechner nicht mit unnötig vielen Prozessen belastet wird, andererseits verhindert dies z.B. die einfache Parallelisierung von parallel verlaufenden Zweigen des Netzwerkes auf Rechnern mit mehr als einem Prozessor. Der Datenaustausch zwischen solchen Modulen ist natürlich einfach durch den gemeinsamen Adreßbereich zu bewerkstelligen. Module, die auf demselben Rechner aber in verschiedenen Prozessen laufen, tauschen ihre Daten über Shared Memory aus: der schreibende Modul legt seine Ausgabedaten dort ab, übermittelt dem lesenden Modul die Adresse der Daten dort, woraufhin dieser die Daten direkt für seine Arbeit nutzen kann. Im Gegensatz zu früheren Versionen von AVS werden, auch wenn beide Module auf einem anderen Rechner als der Flow Executive laufen, keine Daten über das Netzwerk verschickt. Dabei wird unter Umständen auch eine direkte Verbindung zwischen den Modulen zum Datentransport genutzt.

Wenn die beiden Module auf verschiedenen Rechnern laufen, fällt das Shared Memory als direktes Kommunikationsmittel natürlich aus. Hier arbeitet AVS wie folgt (siehe Abbildung 3-2): die Ausgabedaten des Moduls auf dem entfernten Rechner werden durch die AVS Remote Module Support Library und gegebenenfalls nach Konvertierung durch xdr-Routinen über Berkeley Sockets zum lokalen Rechner überspielt. Dort werden sie nötigenfalls für das lokale Datenformat durch xdr-Routinen konvertiert und dann vom Flow Executive an das Filtermodul weitergereicht. Für die Kommunikation über diese Socket-Verbindung werden Remote Procedure Calls verwendet.

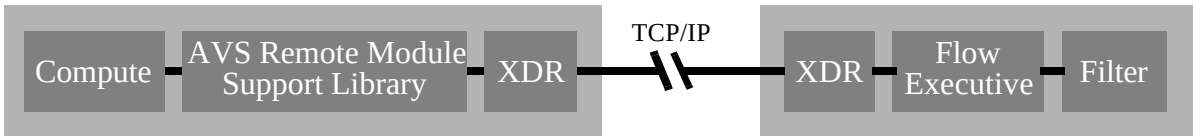

Abbildung 3-2: Datentransfer bei AVS über Rechnergrenzen hinweg

Auf diesem Weg sind etliche teure Zwischenschritte durchzuführen. Nachdem das Modul seine Ausgabedaten erzeugt hat, reicht es diese an die AVS Remote Module Support Library weiter, die wiederum die xdr-Routinen zur Konversion aufruft. Diese Konversion ist natürlich nicht nötig, wenn die beiden Rechner dasselbe Datenformat benutzen. In einer Supercomputer-Workstation Umgebung ist dies jedoch häufig nicht der Fall (insbesondere wenn Cray Computer benutzt werden). Die xdr-Routinen auf Supercomputern sind jedoch höchst ineffizient, verglichen mit den Routinen, die die Hersteller selbst zur Verfügung stellen und die zum Beispiel optimiert sind für die Ausführung auf Vektorprozessoren (siehe hierzu auch Kapitel 5.2, Messungen). Hinzukommt, daß für den Fall, daß der Flow Executive und das Modul nicht in einem einzigen Prozeß untergebracht sind, dort noch einmal die Daten über Sockets vom Flow Executive zum Modul geschickt werden müssen. Darüberhinaus besteht in AVS nicht die Möglich-

keit, Parameter der Netzwerkverbindung wie z.B. die Größe von Socket-Puffern zu verändern. Dies ist jedoch unerläßlich, um in einer Supercomputerumgebung die gebotenen Möglichkeiten auch effizient nutzen zu können (siehe dazu auch [77] und [22]).

Es wird deutlich, daß AVS in seiner gegenwärtigen Form nicht sehr gut für eine mehr als nur einfach verteilte Visualisierungsanwendung (d.h. mit nur einem nicht-lokalen Modul) geeignet ist. Dies zeigt sich insbesondere, wenn zwei Module dieselben Ausgabedaten eines Moduls, das auf einem anderen Rechner läuft, benötigen. In diesem Fall wird nämlich eine Kopie der Ausgabedaten zu jedem Modul geschickt, d.h. die Ausgabedaten werden doppelt auf dem lokalen Rechner angelegt. Zusätzlich zur doppelten Netzauslastung beim Datentransfer werden also auch die Speicherkapazitäten doppelt belastet.

Die Daten, die während der Ausführung eines AVS-Netzwerkes entstehen, sind nur temporärer Natur. Das heißt der Benutzer hat keine Möglichkeit, sich diese Daten anzusehen oder sie zu modifizieren. Daher ist ein Datenmanagement durch den Benutzer auf den Dateibereich im Vorfeld der eigentlichen Visualisierung eingeschränkt; innerhalb des AVS Systems kann man keinen Einfluß auf die Daten nehmen. Über eine SQL-Kommunikationsbibliothek wird dem Programmierer noch die Möglichkeit geboten, mit einem lokalen oder verteilten Datenbank Server zu kommunizieren. Somit arbeitet AVS als Frontend zu einem relationalen Datenbanksystem. Dies ist aber eine sehr lose Kopplung, die man nicht mit einem integrierten Datenmanagement vergleichen kann.

3.1.2 AVS/Express

AVS/Express ist weder ein Abkömmling von AVS noch eine Weiterentwicklung. Vielmehr haben die Entwickler von AVS sich damit eine Basis geschaffen, um objekt-orientierte Ansätze besser umsetzen zu können. AVS/Express ist im wesentlichen ein objekt-orientiertes System, allerdings nicht im Sinne der reinen Lehre, sondern mit einigen Modifikationen [4]. Im Gegensatz zu klassischen Objekten, deren Interface in der Regel aus einem Satz von Methoden besteht, besteht ein Objekt bei AVS/Express aus einer Liste von Parametern, die mit Namen bezeichnet sind. Diese Parameter können einfache Elemente (wie z.B. drei Fließkommazahlen zur Beschreibung einer Farbe oder einer ganzen Zahl für die Höhe eines Widgets) oder aber auch zusammengesetzte Objekte (wie z.B. einem Bild, bestehende aus Breite, Höhe und einem Feld von Bilddaten) sein.

Die Parameter eines Objektes können mit anderen Parametern verbunden werden; diese Verbindung steht dann für ein Teilen der Information zwischen Objekten. Wie in anderen objekt-orientierten Systemen auch, besitzen Objekte Methoden, die jedoch nicht direkt zugreifbar sind. Stattdessen werden diese Methoden automatisch jedesmal aufgerufen, wenn sich ein Parameter ändert. In AVS/Express nennt sich das *Execution Encapsulation.* Parameter eines Moduls, die vom Programmierer dazu bestimmt sind, mit anderen Parametern verbunden werden zu können, bezeichnet man als Ports.

Vergleicht man diese Art, Objekte zu handhaben mit den Modulen aus AVS (siehe Kapitel 3.1.1), so erkennt man, daß es sich im Kern um das gleiche Prinzip handelt: man hat ein Modul, das eine bestimmte Aufgabe erfüllt und dazu mit Eingabedaten versorgt wird und Ausgabedaten erzeugt. Allerdings ist der Objekt-orientierte Ansatz von AVS/Express sehr viel flexibler.

Interessant ist bei AVS/Express der Umgang mit den Daten, die zwischen den Objekten geteilt werden. In welche Richtung eine Änderung eines Datums propagiert wird, hängt davon ab, ob dieses Datum über den Eingabe- oder den Ausgabeport verbunden wird. Falls dies über den Eingabeport geschieht, so wird das Datum innerhalb des Objektes als Referenz auf das Datum geführt, dessen Ausgabeport am anderen Ende der Verbindung liegt. Das Erzeugen von Schleifen durch den Benutzer wird dabei durch

eine Überprüfung im Moment der Verbindungserstellung verhindert. Der Benutzer kann Referenzen sowohl mit der Maus, als auch direkt über die Tastatur eingeben.

Sehr flexibel wird dieser Ansatz dadurch, daß man beliebige Objekte aus den Grundelementen wie z.B. Integer, Short, Float, etc. in sogenannten Modulen zusammensetzen kann. Für jedes dieser Grundelemente läßt sich dann einstellen, ob es als Ein- oder Ausgabeport von außen erreichbar sein soll. Aus Modulen können wiederum weitere Module zusammengesetzt werden, so daß sich eine ganze Hierarchie von gekapselten Objekten erstellen läßt, von der für den Benutzer nur das sichtbar ist, was tatsächlich benötigt wird, um die gewünschte Funktionalität zu erhalten.

So flexibel und konsequent dieses objekt-orientierte Konzept realisiert ist, so fehlt doch in der momentan vorliegenden Version 2.0 die Möglichkeit, die Ausführung von Methoden auf verschiedene Rechner zu verteilen, völlig. Zwar spricht nichts dagegen, daß ein Programmierer Module schreibt, die die Kommunikation mit anderen Rechnern selbst vornehmen, allerdings gehen dabei alle Vorteile des AVS/Express Konzeptes verloren. Diese Tatsache ist natürlich auch der Firma AVS nicht verborgen geblieben, insbesondere, da ja auf AVS/Express die zukünftigen AVS Versionen aufsetzen sollen. Aus diesem Grund ist das Projekt MP/Express initiiert worden.

In diesem Projekt sollte AVS/Express so verbessert werden, daß es möglich wird, verteilte Visualisierungssoftware zu erstellen (siehe dazu [43]). Hauptzielrechner hierfür sollten Workstation-Cluster, symmetrische Multi-Prozessor Systeme und Massiv Parallele Rechner sein. Aus den bisher vorliegenden Unterlagen ist allerdings nicht erkennbar, wie die Verteilung der Daten gehandhabt werden sollte. Der Objektmanager von AVS/Express verwaltet zwar die Objekte, aber aufgrund des bisher monolithischen Ansatzes (kein eigener Prozeß) läßt sich nicht absehen, wie dieser Objektmanager in einer verteilten Umgebung arbeiten wird. Das Projekt MP/Express wird zur Zeit von AVS nicht weiter verfolgt.

Obwohl AVS/Express mit einem Objektmanager arbeitet und identifizierbare Objekte kennt, kann man nicht von einem datenbankorientierten Ansatz sprechen. Die Objekte werden dem Benutzer auf der Benutzungsoberfläche angeboten und lassen sich dort auch bearbeiten, aber es gibt keine Datenbankähnlichen Zugriffsmöglichkeiten. Beim Programmieren von Modulen hat man allerdings die Möglichkeit, eine einfache Suche anhand des Namens eines Objektes durchzuführen.

3.1.3 IRIS-Explorer

Die Systemarchitektur von IRIS-Explorer wurde von Craig Upson, dem Entwickler von AVS, konzipiert. Die Vermutung, daß auf den Erfahrungen mit der Entwicklung von AVS aufbauend einige Veränderungen oder Verbesserungen in das Design eingebaut wurden, bestätigen sich bei einem Blick auf die Systemarchitektur von IRIS-Explorer (siehe Abbildung 3-3) [19] [26] [27].

Im Gegensatz zu AVS gibt es hier nicht nur einen Flow Executive, der alle Abläufe koordiniert, sondern ein zweistufiges Konzept. Dem Flow Executive entspricht dabei im wesentlichen der Global Controller (GC). Hier ist die Information über die Topologie gespeichert (welcher Modul wo läuft und in welcher Reihenfolge die Module gestartet werden sollen). Auf jedem beteiligten Rechner läuft zusätzlich ein sogenannter Local Controller (LC), der für die Ausführung der Module sowie das Herstellen der Sokket-Verbindungen zwischen ihnen, die Nutzung von UNIX-Pipes und das Shared Memory zuständig ist. Durch diesen Ansatz soll die Kommunikation zwischen den Rechnern auf ein Minimum beschränkt werden. Das heißt, die Steuerung der Module D und E findet lokal auf der entfernten Workstation statt, ohne daß der Global Controller diese Aktion überwachen muß. Das Userinterface (UI) ist direkt mit dem Global Controller verbunden.

Dies läßt sich als Anzeichen dafür werten, daß bei IRIS-Explorer im Gegensatz zu AVS die Verteilung der Module über verschiedene Rechner von Anfang an bei der Konzeption berücksichtigt wurde. Dazu gehört auch, daß die Verwendung von Shared Memory für Daten, die auf demselben Rechner weiterverarbeitet werden, von vornherein implementiert wurde. Allerdings wurde dies nicht immer für alle Datentypen unterstützt. Erst seit Version 3.0 werden zum Beispiel auch Geometriedaten zwischen Modulen über Shared Memory ausgetauscht.

Der Zugriff auf die Daten, die zwischen den Modulen ausgetauscht werden, ist auch beim IRIS-Explorer nicht direkt möglich. Der Benutzer hat keinerlei Möglichkeit, sich ein solches Datenobjekt anzusehen oder zu ändern. Es gibt auch keine Übersicht über die existierenden Datenobjekte. Weder der globale noch der lokale Controller haben eine Datenmanagementkomponente.

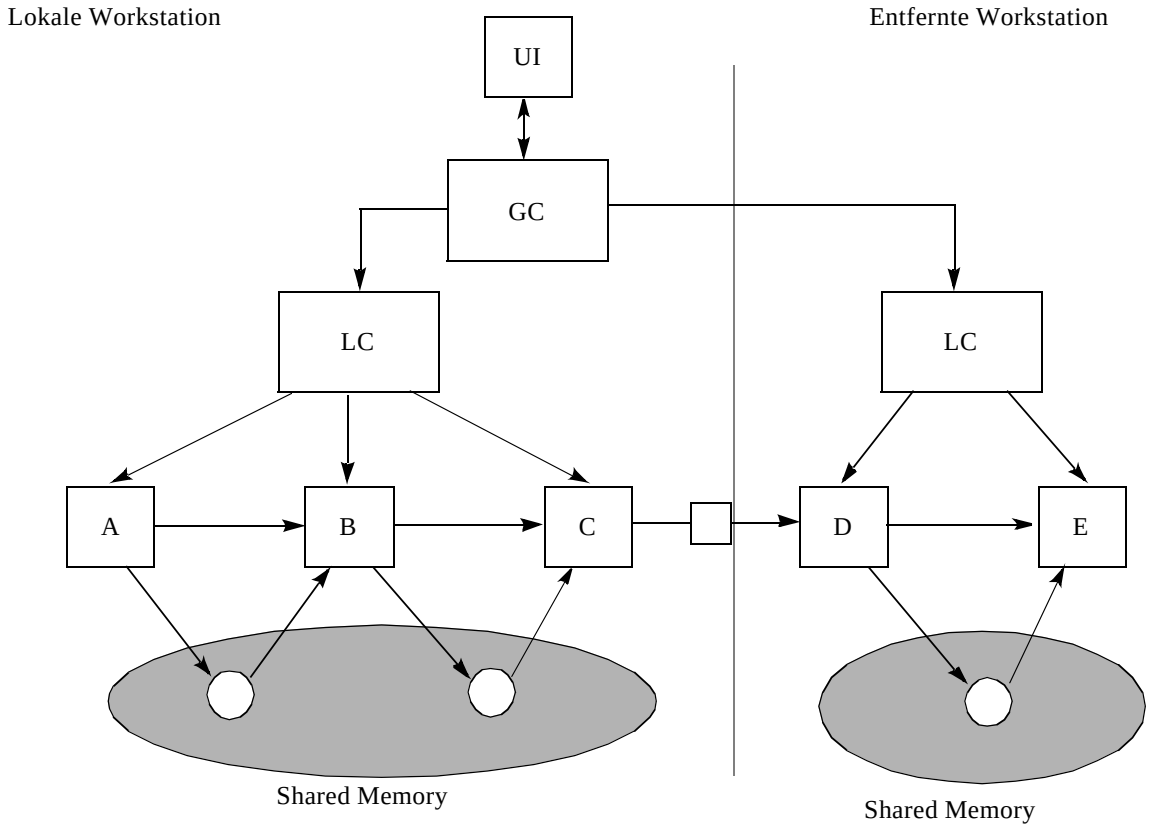

Abbildung 3-3: Die Systemarchitektur des IRIS-Explorer

3.1.4 Data-Explorer

Der IBM Data Explorer folgt zwar auch dem Paradigma des visuellen Programmierens, aber seine Prozeßstruktur ist monolithisch. Das heißt, daß einzelne Module (oder auch Gruppen von Modulen) nicht jeweils einem Prozeß entsprechen, sondern vielmehr alle Module in einem einzigen Executable gelinkt sind. Die Möglichkeit, einzelne Module auf andere Rechner auszulagern war daher nicht von Anfang an gegeben, sondern wurde erst in einer späteren Version hinzugefügt, allerdings ohne die Grundarchitektur zu ändern [1] [72].

In dieser Grundarchitektur (siehe Abbildung 3-4) gibt es nur zwei Prozesse: einen Client und einen Server (der Executive). Der Client ist das Frontend, über das der Benutzer die Anwendung steuert, im

Server wird die tatsächliche Visualisierungsarbeit geleistet und die gesamte Anwendung gesteuert. Beide Prozesse können auch auf verschiedenen Rechnern laufen, so daß z.B. die Rechenkapazität eines Computers ohne Graphikhardware genutzt werden kann und die Bilder anschließend auf der Workstations des Benutzers dargestellt werden. In diesem Fall wird die Graphikdarstellung über X-Windows abgewikkelt; befinden sich Server und Client auf demselben Rechner, kann eine eventuell vorhandene Graphikhardware für die Darstellung genutzt werden.

Sollen Module auch auf einem anderen Rechner laufen, kann man sogenannte Execution Groups bilden. In solchen Gruppen werden mehrere Module zusammengefaßt, die dann einem speziellen Rechner zugewiesen werden können. Beim Start des Data Explorers wird dann auf jedem der beteiligten Rechner ein Executive gestartet, der für die Ausführung der Module seiner Execution Group zuständig ist. Der Master Executive (das ist der, mit dem das Benutzer-Interface verbunden ist) initialisiert die Kommunikation mit den anderen Executives und steuert die Ausführung der Programme.

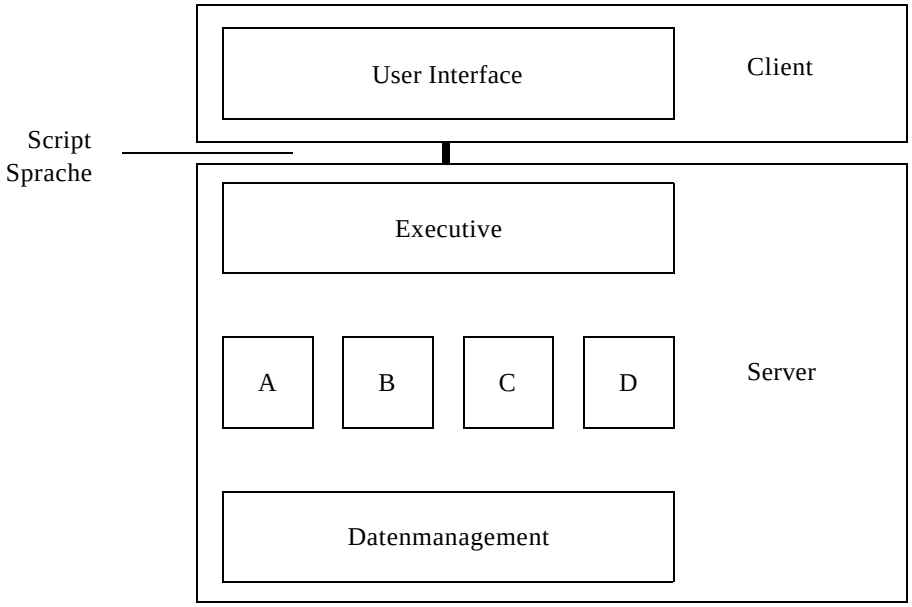

Abbildung 3-4: Kommunikation des IBM Data Explorer

Eine andere Möglichkeit, eine Anwendung zu verteilen, sind die Outboard Module. Outboard Module werden separat übersetzt und laufen auch als separater Prozeß; die Datenkommunikation wird hier über Sockets abgewickelt, selbst wenn die kommunizierenden Prozesse auf demselben Computer laufen. Ein Outboard Modul kann auch auf einem anderen Computer laufen; dann wird eine Socket Verbindung zum Haupt-Executive aufgebaut. Die Autorisierung bei der Ausführung eines Moduls auf einem anderen Computer wird über den .rhosts Mechanismus sichergestellt. Dies ist jedoch eine relativ unsichere Methode und wird daher zum Beispiel auf den großen Compute-Servern des Rechenzentrums der Universität Stuttgart nicht unterstützt.

3.1.5 Ensight

Wie bereits in der Einleitung zu diesem Kapitel erwähnt wurde, folgt Ensight nicht dem Paradigma des visuellen Programmierens [15]. Damit zusammenhängend findet sich auch der Modul-orientierte Ansatz nicht bei Ensight. Vielmehr ist Ensight (das als MPGS ursprünglich von der Firma Cray Research Inc. entwickelt wurde) ein klares Client/Server-Programm. Die Ausführung der Visualisierungsalgorith-

men findet auf dem Server statt (der früher in der Regel ein Cray Supercomputer war) und die Benutzungsschnittstelle sowie die Graphikausgabe finden sich auf dem Client (normalerweise die Workstation am Arbeitsplatz des Benutzers).

Die Daten, die dargestellt werden, stammen in der Regel aus Dateien. Aufbauend auf der Struktur dieser Daten kann der Benutzer verschiedene Algorithmen auf diese Daten anwenden (z.B. eine Schnittfläche durchlegen oder Partikel einstreuen). Hier wird eine quasi-natürliche Objekt-Orientierung aus der Sicht des Nutzers angewendet. Allerdings besteht keine weitergehende Kontrolle über diese Objekte oder die Resultate aus der Anwendung der Algorithmen, wie sie bei einer Datenverwaltung vorhanden wäre. Die einzige Möglichkeit, eine Visualisierung abzuspeichern, besteht darin, neben dem ursprünglichen Datensatz eine sogenannte View zu schreiben, die sämtliche Parameter, die für diese Visualisierung notwendig sind, enthält.

Da Ensight als MPGS (Multi Purpose Graphics System) ursprünglich von Cray Research Inc. entwikkelt wurde, finden sich viele speziell auf die Nutzung von Supercomputern optimierte Eigenschaften. So ist die Cray-Server Version beispielsweise vektorisiert und die Kommunikation erfolgt nicht wie bei AVS über die recht langsamen xdr-Funktionen.

3.2 Übersichtstabelle

In der folgenden Tabelle sind die wesentlichen Merkmale, die an ein verteiltes Visualisierungssystem gestellt werden, für die Systeme aus Kapitel 2 und Kapitel 3 aufgelistet. Die Definition dieser Merkmale lautet dabei:

Tabelle 3-1: Definition der Vergleichskriterien

Objekt-Orientierung	das System folgt einem objekt-orientierten Ansatz im generellen Sinne
Klassen	es existieren verschiedene Klassen für die verschiedenen Datentypen
Vererbung	es gibt eine Möglichkeit, Klassen abzuleiten
Methoden	es gibt Methoden, die zu den jeweiligen Klassen gehören
Module	es gibt Module, die über wohldefinierte Schnittstellen Daten miteinander austauschen (eine einfache Art von Objekt-Orientierung)
Verteilung	verschiedene Methoden können beliebig auf verschiedenen Rechnern ausgeführt werden
Datenverwaltung	es gibt eine dedizierte Datenverwaltung
Objektzugriff	es gibt eine Möglichkeit auf einzelne Datenobjekte zuzugreifen
Objektverteilung	Objekte werden automatisch auf die Rechner gebracht, auf denen sie benötigt werden
Supercomputer	die Software ist auf Supercomputern verfügbar
Optimierung	die Software ist für Supercomputer optimiert

Die Informationen die in der folgenden Tabelle dargestellt sind, beziehen sich auf die neueste Version der entsprechenden Software. Da nicht bei allen Herstellern die Dokumentation ausreichend tiefen Einblick in die entsprechenden Interna der Software gibt, findet sich in den Feldern, zu denen keine hinreichenden Informationen gefunden wurden ein *; - bedeutet nicht anwendbar.

Tabelle 3-2: Vergleich der Visualisierungssysteme

Software / Eigenschaft	COVISE	GRAPE	RSYST	AVS	AVS/ Express	Ensight	Iris Explorer	IBM Data Explorer
Objekt-Orientierung	Ja	Ja	Ja	Nein	Ja	Nein	Nein	Nein
Klassen	Ja	Ja	Ja	-	Ja	-	-	-
Vererbung	Ja	Ja	Nein	-	Ja	-	-	-
Methoden	Ja	Ja	Ja	-	Ja	-	-	-
Module	Ja	Nein	Ja	Ja	Ja	Nein	Ja	Ja
Verteilung	Ja	Nein	Nein	Ja	Nein	Nein[1)]	Ja	Ja
Datenverwaltung	Ja	Nein	Ja	Nein	Ja	Nein	Nein	Nein
Objektzugriff	Ja	-	Ja	-	Ja	-	-	-
Objektverteilung	Ja	-	Ja	-	Nein	-	-	-
Supercomputer	Ja	Nein	Ja	Ja	Nein	Ja	Ja	Ja
Optimierung	Ja	-	Ja	Nein	Nein	Ja	*	*

1) Bei Ensight werden immer zwei Prozesse gestartet: ein Server, der die berechnungsintensiven Aufgaben ausführt, und ein Client, der das User-Interface und die Graphikausgabe handhabt. Die einzige Variationsmöglichkeit besteht darin, diese beiden Prozesse entweder auf demselben oder auf zwei verschiedenen Rechnern laufen zu lassen.

Kapitel 4

Konzept eines verteilten Visualisierungssystems

Nach ausführlicher Untersuchung der bislang verfügbaren Visualisierungssysteme wurde ein Konzept entwickelt, das die Vorteile dieser Systeme vereinen und die Nachteile vermeiden soll. Besondere Aufmerksamkeit wurde dabei von Anfang an der Verteilung der einzelnen Komponenten auf verschiedene Rechner gewidmet.

In Kapitel 2 wurde die System-Architektur bereits kurz vorgestellt. Dieses Kapitel beginnt mit einer detaillierten Darstellung der verteilten Datenverwaltung (4.1). Des weiteren werden die Datenobjekte vorgestellt (4.2, dabei wird auch auf den Aspekt der Objektorientierung eingegangen) und damit zusammenhängend Persistenz (4.3) und partitionierte Objekte (4.4) dargestellt. Abschließend wird ein Blick auf die besonderen Anstrengungen die zur Nutzung von High-Performance-Infrastruktur unternommen wurden (4.5) geworfen und die wesentlichen Strategien bei der Implementierung werden erläutert (4.6).

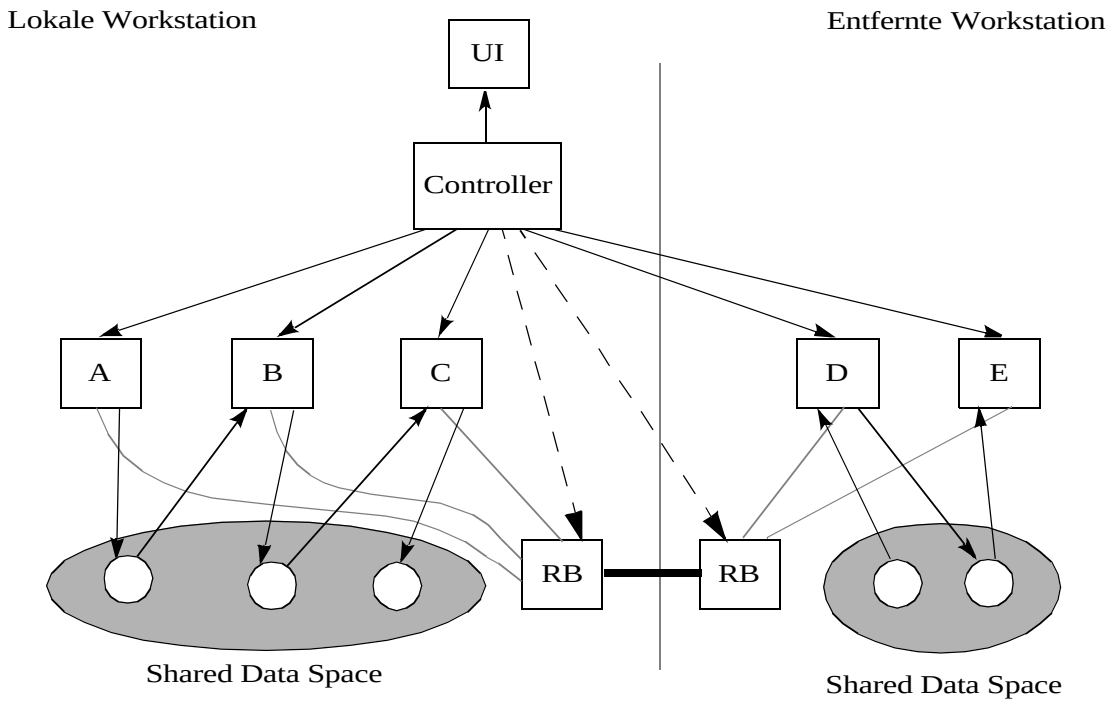

Abbildung 4-1: Die COVISE Systemarchitektur

4.1 Die verteilte Datenverwaltung

4.1.1 Der Datenaustausch zwischen Modulen

Im Gegensatz zu AVS und IRIS Explorer sind die Module bei COVISE nicht direkt miteinander verbunden (vergleiche dazu die Abbildungen 3-1, 3-3 und 4-1). Während bei AVS anfänglich die Daten zwischen aufeinanderfolgenden Modulen immer mittels TCP-Sockets übertragen wurden, wurde dies bei IRIS-Explorer von Beginn an nur bei Modulen, die auf verschiedenen Rechnern laufen, getan. In späteren AVS-Versionen wurde, ebenso wie bei IRIS Explorer, für den Austausch von Daten zwischen lokalen Modulen Shared Memory genutzt. Dennoch sind die Module nach wie vor mittels eines TCP-Sockets miteinander verbunden. Sinn dieser Verbindung ist, die Information über Daten direkt von einem Modul zum nächsten fließen zu lassen.

In COVISE gibt es keine direkte Verbindung zwischen den Modulen. Ein Modul weiß nie (und muß auch nicht wissen), welche und wieviele Module vor oder nach ihm laufen und ob diese lokal oder verteilt sind. Der Zugriff auf Daten erfolgt immer über den Shared Data Space, und die Information, wo die gewünschten Daten zu finden sind, wird beim Request Broker eingeholt. Wesentlich bei der Realisierung dieses Datenaustauschs ist, daß es sich bei den Daten um identifizierbare Objekte handelt. Die Identifizierung im Rahmen der realisierten COVISE-Software erfolgt durch einen Namen, allerdings besitzt jedes Objekt eine eindeutige Objekt-Id.

Lokaler Datenaustausch

Nun soll der Ablauf der Datenkommunikation zwischen zwei aufeinanderfolgenden Modulen während der Ausführung einer Applikation an einem Beispiel erläutert werden. Wir betrachten in diesem Szenario einen Lesemodul (A in Abbildung 4-2), der Daten aus einer Datei einliest und einen darauffolgenden Filtermodul (B).

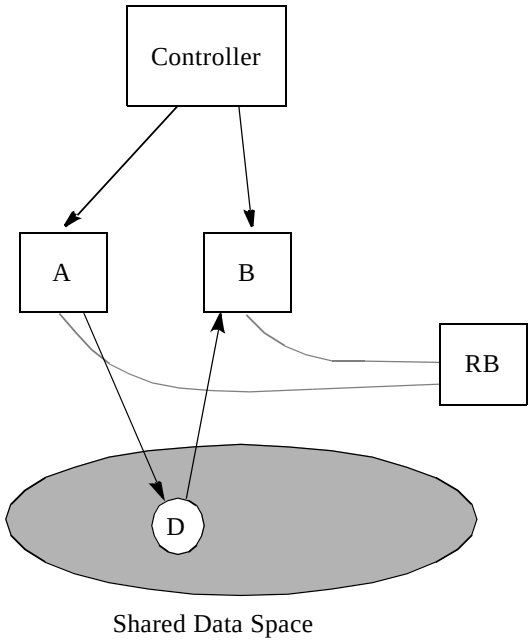

Abbildung 4-2: Lokaler Datenaustausch

Wenn der Controller die Startmessage an den Lesemodul sendet, werden dieser die Namen, unter denen der Lesemodul seine Datenobjekte ablegen soll, mitgegeben. In diesem Beispiel werde nur ein Datenobjekt ausgetauscht. Der Controller hat vollständige Kenntnis über die gesamten Parameter des Lesemoduls; einzelne Parameter wie z.B. der Dateiname werden über die Benutzungsschnittstelle vom Benutzer abgefragt. Daraufhin beginnt der Lesemodul mit seiner Arbeit. Sowie sich ergibt, wieviel Speicherplatz benötigt wird, um die gelesenen Daten im Shared Data Space abzulegen, erzeugt der Lesemodul ein Datenobjekt des benötigten Typs in der benötigten Größe. Dazu werden die Größen der gewünschten Datenbereiche sowie die Struktur des Datenobjektes in den Grunddatentypen und der Name des zu erzeugenden Datenobjekts an den Request Broker übermittelt. Dieser stellt sicher, daß kein gleichnamiges Datenobjekt existiert, sucht dann in seinem Shared Data Space nach entsprechenden Bereichen, setzt das Datenobjekt im Shared Data Space daraus zusammen und sendet anschließend dessen Adresse in einem prozeß-unabhängigen Format an den Lesemodul zurück. Mit Hilfe dieser Information wird dann ein lokales Datenobjekt erzeugt, über dessen Methoden der Zugriff des Lesemoduls auf die Daten im Shared Data Space erfolgt. Nun kann der Lesemodul die gelesenen Daten in den Shared Data Space schreiben und abschließend eine Finishmessage an den Controller zurückschicken.

Der Controller sendet nun an den darauffolgenden Filtermodul wiederum eine Startmessage, die neben den modulspezifischen Parametern den Namen des Datenobjektes enthält, das der Filtermodul als Eingabedaten benötigt. Dieser erzeugt nun ein lokales Datenobjekt, das ihm den Zugriff auf die Daten im Shared Data Space ermöglicht. Dazu wird einfach der Name des gewünschten Datenobjektes an den Request Broker geschickt, dieser sucht die Adresse des dazugehörigen Speicherbereiches im Shared Data Space und schickt diese an den Filtermodul zurück. Das lokale Datenobjekt kann nun seine Zugriffsmethoden so initialisieren, daß die richtigen Speicherbereiche angesprochen werden. Der Filtermodul kann daraufhin mit seiner eigentlichen Arbeit beginnen.

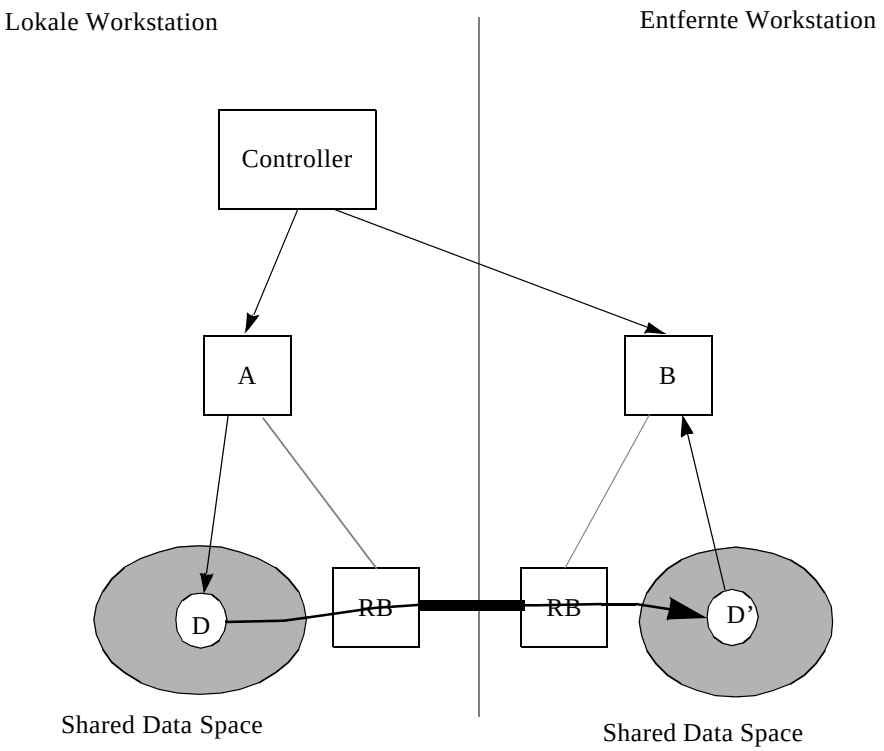

Abbildung 4-3: Nicht-Lokaler Datenaustausch

Nicht-Lokaler Datenaustausch

Nun soll der Fall betrachtet werden, daß das Lese- und das Filtermodul nicht auf demselben Rechner ausgeführt werden (siehe Abbildung 4-3). Zunächst läuft alles nach genau demselben Schema ab wie im lokalen Fall. Der Lesemodul bekommt mit der Startmessage den Namen des Objektes, besorgt sich bei seinem Request Broker den nötigen Speicherbereich im Shared Data Space, füllt ihn mit seinen Daten und schickt eine Finishmessage an den Controller zurück. Daraufhin wird eine Startmessage mit dem Objektnamen an den Filtermodul geschickt, woraufhin dieser seinem Request Broker mitteilt, daß er auf dieses Objekt zugreifen möchte. Dieser Request Broker findet jedoch in dem von ihm verwalteten Shared Data Space kein Datenobjekt dieses Namens. Daraufhin setzt er sich mit dem direkt mit ihm verbundenen Request Broker in Verbindung und bittet diesen, nachzuschauen, ob er ein solches Datenobjekt in seinem Shared Data Space hat und es gegebenenfalls zu schicken. Nachdem nun eine Kopie D' des Datenobjektes D angelegt worden ist, wird die Adresse dieser Kopie an den Filtermodul B zurückgeschickt. Dieser kann daraufhin wieder ganz wie im lokalen Fall auf das Datenobjekt zugreifen und seine Arbeit aufnehmen.

Die Verbindung zwischen den Request Brokern

Im Vergleich zu den Verbindungen zwischen den anderen Prozessen kommt der Verbindung zwischen zwei Request Brokern eine spezielle Bedeutung zu. Die Tatsache, daß der gesamte Datenaustausch zwischen zwei Rechnern genau über diese Verbindung geht, macht es besonders leicht, Optimierungen durchzuführen. (Natürlich gehen auch über die Verbindungen zwischen dem Controller und den auf andere Rechner verteilten Modulen Daten; da es sich aber um Steuerinformationen im Umfang von nur wenigen 100 oder 1000 Byte handelt, können diese vernachlässigt werden im Vergleich zu den möglicherweise mehreren Megabyte, die zwischen Request Brokern ausgetauscht werden.)

Leider sind noch nicht alle Rechnerhersteller dazu übergegangen, IEEE Datentypen zu verwenden. Auf den Cray Vektorsupercomputern z.B. wird ein Cray-spezifisches Floating-Point-Format benutzt, das, wenn die Daten auf Workstations weiterverarbeitet werden sollen, in das IEEE Format umgewandelt werden muß. Diese Umwandlung gehört zu den Aufgaben der Request Broker. Beim ersten Aufbau der Verbindung zwischen zwei Request Brokern wird überprüft, ob eine Konversion nötig ist. In diesem (und nur in diesem) Fall werden beim nötigen Datentransfer die Daten konvertiert und zwar auf dem Rechner, der dafür am besten geeignet ist (Messungen haben gezeigt, daß der Durchsatz einer Cray C90 beim Konvertieren deutlich größer ist als der einer SGI Indigo R4000 beim Kopieren!). Um eine effiziente Konvertierung vornehmen zu können, sind natürlich Zusatzinformationen über Typ und Feldgrößen im Shared Data Space mitgespeichert (näheres dazu in Abschnitt 4.2).

Verschiedene Netzwerktechnologien erfordern unter Umständen auch verschiedene Parameter für die Verbindung, um den optimalen Gebrauch der darunterliegenden Hardware zu ermöglichen. So hat zum Beispiel die Einstellung der Puffergrößen bei TCP-Sockets einen ganz wesentlichen Einfluß auf den erreichbaren Durchsatz. Die Konzentration der Datenverbindungen auf die Request Broker ermöglicht es, eine auf den für die jeweilige Hardwaresituation abgestimmte Umgebung zu erzeugen. Das könnte theoretisch soweit gehen, daß auf jedem Rechner in einem Pool eine für dessen Netzwerkinterfaces optimierte Version des Request Brokers vorhanden ist, aber dieselben Module zur Ausführung kommen. Das mag sich zwar etwas übertrieben anhören, doch unsere Erfahrungen bei den Messungen (siehe Abschnitt 5.2) haben gezeigt, daß gerade bei sehr schnellen Technologien (wie z.B. HIPPI) bereits das Abweichen eines Parameters vom Optimum zu deutlichen Leistungseinbußen führt.

4.1.2 Vorteile der fehlenden Direktverbindung

Die Vermeidung einer direkten Verbindung zwischen den Modulen hat mehrere Vorteile. Die Programmierung eines Moduls wird von viel Ballast befreit. Ein Modul empfängt seine Befehle (start, stop, beenden, etc.) und Informationen (z.B. Parameter) immer über dieselbe Verbindung vom Controller, alles was mit dem Datenaustausch mit anderen Modulen zu tun hat, geht über die Verbindung zum Request Broker. Auch ist es für einen Modul vollkommen transparent, ob der nächste Modul lokal oder auf einem anderen Rechner liegt. Der Austausch von Daten mit anderen Modulen wird immer auf dieselbe quasi-lokale Weise durchgeführt. Das Moduldesign wird dadurch wesentlich vereinfacht, was sich bei der Entwicklung neuer Module deutlich bemerkbar macht. Der ganze Aufwand, der zur Verwaltung mehrerer Socketverbindungen und zur internen Ablaufsteuerung nötig ist, entfällt.

Weitere Vorteile bestehen in der weitergehenden Trennung von anwendungsspezifischen und kommunikationsspezifischen Systemteilen (wenn man einmal von der Mindestkommunikation eines Moduls mit dem Controller und dem Request Broker absieht). Dadurch, daß die komplette Kommunikation von Daten (im Gegensatz zur Steuerinformation) an einer wohldefinierten Stelle im System stattfindet, ist es besonders einfach, die zur optimalen Nutzung von High-Performance-Hardware nötigen Anpassungen und Parametermodifikationen schnell und effizient durchzuführen, ohne auch nur einen Modul neu kompilieren zu müssen.

Dies ist insbesondere im Hinblick auf die Erweiterung eines Hardwaresystems durch neue Netzwerkkomponenten wichtig. Um eine neue Netztechnologie, die besondere Schnittstellen erfordert (wie z.B. Quality of Service bei ATM) zu nutzen, müssen diese Schnittstellen nur im Request Broker integriert werden. Alle Module sind weiterhin unverändert lauffähig und können sofort von den Vorteilen der neuen Technologie profitieren.

4.1.3 Mechanismen zur Konfliktvermeidung

Die einfache Tatsache, daß auf ein Datenobjekt von mehreren Modulen in schreibender oder lesender Weise zugegriffen werden kann, birgt bereits eine gewisse Gefahr in sich: während ein Modul liest, könnte ein zweiter Modul schreiben, so daß entweder ein gar nicht mehr aktuelles Objekt gelesen wird oder aber (was in der Regel deutlich schlimmer ist) ein inkonsistentes Bild als Ergebnis des Lesevorganges entsteht (eine Hälfte des Datenobjekts war während des Lesens noch auf dem alten Stand, die andere Hälfte bereits durch das neue Datenobjekt ersetzt). Durch die Verteilung von Datenobjekten auf mehrere Rechner gewinnt diese Problematik noch weiter an Bedeutung.

Vereinfachend wiederum wirkt, daß es sich bei diesem System nicht um ein voll ausgebautes Datenbanksystem handelt, daß sich mit vielen gleichzeitigen Schreib- und Lesezugriffen auf dieselben Datensätze innerhalb kürzester Zeit auseinandersetzen muß. Vielmehr werden die zeitkritischen Zugriffe in der Regel durch den Ablauf einer mittels des visuellen Programmierens zusammengesetzten Anwendung bestimmt. Dadurch wird zum Beispiel nie von mehr als einem Modul gleichzeitig schreibend auf ein Datenobjekt zugegriffen. Weiterhin erfolgt eine Änderung eines Datenobjektes in der Regel durch den Modul, der es auch erzeugt hat. Erfolgen Zugriffe nicht nach diesem Schema, so sind sie durch den Benutzer direkt ausgelöst und unterliegen daher auch nicht den strengen Performance-orientierten Ansprüchen einer automatischen Ausführung: für einen menschlichen Benutzer sind Antwortzeiten im Millisekundenbereich durchaus akzeptabel; wenn sich bei der Ausführung vieler Module solche Verzögerungen im Millisekundenbereich jedoch aufsummieren, kann es durchaus zu einer ernsthaften Beeinträchtigung der Performance des Gesamtsystems kommen.

Der lokale Fall

Aus diesem Grunde sind die Mechanismen zur Zugriffssteuerung so ausgelegt, daß sie bei einer automatischen Ausführung am effizientesten funktionieren. Bei einem normalen Durchlauf einer solchen Anwendung wird ein Modul seine (bereits existierenden) Eingabeobjekte lesen, seine Arbeit ausführen und dabei oder danach ein Ausgabeobjekt erzeugen und schreiben. Dementsprechend ist auch die Datenverwaltung im Request Broker aufgebaut: wenn ein Modul nach einem bereits existierenden Datenobjekt fragt, wird ihm automatisch mit der Rücksendung der Adresse das Leserecht erteilt. Nach Beendigung des Lesens muß das Modul das Leserecht wieder zurückgeben. Wenn ein Modul ein Datenobjekt neu anlegt, so wird ihm, nachdem der Speicher reserviert und die Adresse dieses Speicherbereiches zurückgeschickt wurde, automatisch das Schreibrecht zugeteilt. Nachdem das Modul seine Arbeit verrichtet hat (egal ob lesend oder schreibend), gibt er seine Schreib- oder Leserechte an dem Datenobjekt zurück (siehe dazu Kapitel 4.6, Wesentliche Strategien der Implementierung).

Des weiteren gelten einige bewußt einfach gehaltene Regeln für den gleichzeitigen Zugriff:

- solange ein Datenobjekt geschrieben wird, kann es nicht gelesen werden
- solange ein Datenobjekt gelesen wird, kann es nicht geschrieben werden
- die Zahl der gleichzeitig auf ein Datenobjekt schreibenden Module ist auf eins beschränkt
- es können beliebig viele Module gleichzeitig lesen

Theoretisch kann es trotz dieser Regeln durchaus zu einem sogenannten „Deadlock“ kommen: ein Modul (A) benötigt, bevor es seine Ausgabedaten schreiben kann, ein Eingabedatenobjekt, das wiederum von einem anderen Modul (B) geschrieben werden soll, das auf die Ausgabedaten des ersten Module wartet (siehe Abbildung 4-4). In der Praxis kann diese Konstellation jedoch nicht vorkommen, da eine solche Verbindung beim Aufbau einer Anwendung mittels des visuellen Programmierens einfach nicht zugelassen wird. Weiterhin handelt es sich durch die Serialisierung der Aktionen im Controller quasi um eine Ein-Benutzeranwendung, d.h. es kann nicht vorkommen, daß mehrere Benutzer unkontrolliert auf dieselben Daten schreibend zugreifen.

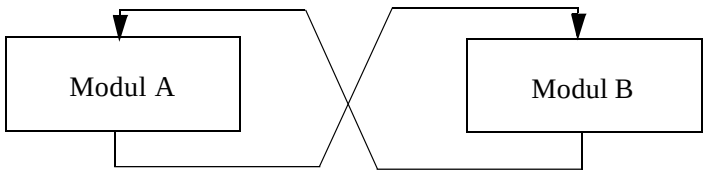

Abbildung 4-4: "Deadlock"-Situation

Nach der Definition des Benutzer-Isolationsgrades von Transaktionen in [21] läßt sich diese Art des Zugriffs mit dem Isolationsgrad 3 belegen. Anhand der dortigen Benutzer-Definition von Isolation soll dies überprüft werden. Eine Transaktion ist in diesem Kontext definiert als ein abgeschlossener Zugriff eines Moduls auf ein Datenobjekt im Shared Data Space.

Eine Transaktion ist isoliert von anderen Transaktionen, wenn gilt:

(0) *Eine Transaktion T überschreibt keine ungültigen Daten anderer Transaktionen*

Gilt, da ein Schreibzugriff nur dann erlaubt wird, wenn kein anderer Zugriff stattfindet

(1) *Von T geschriebene Daten werden vor T's Abschluß von anderen Transaktionen weder gelesen noch geschrieben*

Gilt, da während eines Schreibzugriffes keine anderen Zugriffe (weder lesen noch schreiben) erlaubt werden

(2) *T liest keine ungültigen Daten anderer Transaktionen*

Gilt, da während eines Schreibzugriffes keine anderen Schreibzugriffe erlaubt werden

(3) *Andere Transaktionen schreiben keine (ungültigen) Daten, die T liest, bevor T nicht beendet ist*

Gilt, da während eines Lesezugriffes keine Schreibzugriffe erlaubt werden und ein Lesezugriff auf ein Datenobjekt, auf das bereits schreibend zugegriffen wird, nicht erlaubt wird.

Die Zugriffe von Modulen auf Datenobjekte, die über den Request Broker gesteuert werden, können also im lokalen Fall als sicher gelten.

Der nicht-lokale Fall

Werden Datenobjekte auf einem anderen Rechner erzeugt als auf dem, auf dem sie gelesen werden, so ergibt sich eine zusätzliche Problematik: die Existenz einer Kopie und die Übereinstimmung mit ihrem Original. Beim Standardablauf einer Anwendung wird die Kopie genau dann erzeugt, wenn sie von einem lesenden Modul gebraucht wird, d.h. auch hier ist sichergestellt, daß nicht gleichzeitig gelesen und geschrieben werden muß. Daher wird einfach zum gegebenen Zeitpunkt eine Kopie des Objektes auf dem zweiten Rechner erzeugt.

Um jedoch die volle Isolation auch im Falle mehrerer Request Broker garantieren zu können, müssen die Locking-Mechanismen ausgeweitet werden. Dies läßt sich am einfachsten dadurch erreichen, daß die bislang lokalen Vorgänge auf eine beliebige Anzahl von Request Brokern erweitert werden. D.h. ein Zugriff auf eine Kopie eines Datenobjektes kann erst erfolgen, nachdem der Request Broker der Kopie beim Request Broker des Originals sichergestellt hat, daß der Zugriff erlaubt ist. Dadurch wird praktisch das Modell aus dem lokalen Fall dahingehend erweitert, daß nicht-lokale Zugriffe auf quasi-lokale Zugriffe reduziert werden.

Aus Gründen der Systemperformance erscheint es jedoch sinnvoll, ein wenig von diesem Modell abzuweichen. Dabei ist der Fall, daß ein Datenobjekt erst noch kopiert werden muß, sehr einfach zu handhaben: mit dem Anlegen der Kopie wird sozusagen der Lesezugriff gleich mitkopiert. Da auch mehrere Lesezugriffe gleichzeitig aktiv sein können, können somit auch mehrere Kopien gleichzeitig zum Lesen freigegeben werden. Betrachtet man nun das Anlegen einer Kopie als permanenten Lesezugriff (damit der Request Broker Leseanfragen „seiner" Module ohne zeitraubende Nachfrage genehmigen kann), bleibt nur der Fall eines Schreibzugriffes gesondert zu behandeln.

Dies soll zunächst für den Fall, daß das Original zum Schreibzugriff freigegeben werden soll, betrachtet werden: Hierfür muß jeder Request Broker, der eine Kopie des Datenobjektes besitzt, befragt werden, ob nicht gerade eine Lesezugriff wirklich aktiv ist. Falls ja, kann man entweder warten bis dieser Zugriff beendet ist oder aber den Schreibzugriff ablehnen (und damit das Warten in den Schreibzugriff anfordernden Modul verlagern). Falls kein Lesezugriff bei einem der Request Broker, die eine Kopie haben, aktiv ist, kann der Schreibzugriff erteilt werden. Aus Effizienzgründen wird man bei der Anfrage

bei diesen Request Brokern gleich den Lock für den Schreibzugriff mitsetzen, so daß mit dem Einverständnis des letzten Request Brokers der Schreibzugriff dem anfragenden Modul erteilt werden kann.

Wird nun die Anfrage nach dem Schreibzugriff nicht für das Originaldatenobjekt, sondern für eine Kopie gestellt, so wird der Request Broker der Kopie beim Request Broker des Originals um die Schreibrechte nachsuchen. Dieser wiederum fragt alle anderen Request Broker, die eine Kopie halten, nach dem Schreibrecht. Durch diese Handhabung wird dem üblichen Ablauf einer Anwendung, in der ein Objekt von einem Modul angelegt und anschließend von einem oder mehreren anderen Modulen gelesen wird, Rechnung getragen. Unter diesen Umständen wird eine optimale Performance erreicht, während der wesentlich seltenere Fall, daß ein bereits existierendes Datenobjekt noch einmal zum Schreiben geöffnet wird, in seiner Performance beeinträchtigt wird.

4.2 Die Datenobjekte

Module arbeiten mit zwei Arten von Information: Steuerinformation und auszutauschende Daten. Zu den Steuerinformationen gehören zunächst einmal der Befehl vom Controller, mit der Ausführung zu beginnen. Die meisten Module haben zusätzlich allerdings auch eine ganze Reihe von Parametern, die das Verhalten des Moduls beeinflussen. Das können beim Renderer zum Beispiel die Einstellung bezüglich der Stereodarstellung sein oder bei einem Isoflächenmodul der Wert, zu dem die Isofläche gerechnet werden soll oder ob Normalen generiert werden sollen. All dies sind jedoch Informationen, die sich nur auf das Modul selbst und sein Arbeitsweise beziehen und keine anderen Module beeinflussen.

Alle Informationen, die auch andere Module beeinflussen, d.h. die irgendeinen Informationsaustausch zwischen zwei Modulen betreffen, werden über sogenannte Datenobjekte ausgetauscht. Diese Datenobjekte liegen im Shared Data Space (siehe 4.1.1) und werden vom Request Broker verwaltet. Der Zugriff von Modulen auf diese Datenobjekte erfolgt immer lokal: normalerweise über Shared Memory. Sollte kein Shared Memory auf dem benutzten Rechner verfügbar sein (d.h. nicht implementiert) werden jeweils ein Modul und ein Request Broker in einem Executable integriert und der Shared Data Space liegt auf dem Heap.

4.2.1 Anforderungen an die Speicherung der Datenobjekte

Die Speicherung der Datenobjekte im Shared Data Space muß die folgenden beiden Zugriffsarten ermöglichen:

- ein Modul möchte die Daten in diesem Datenobjekt schreiben oder lesen
- ein Request Broker möchte ein Datenobjekt zum Request Broker auf einem anderen Rechner senden

Aus diesen Zugriffsarten ergeben sich folgende Bedingungen für die Speicherung:

- *Für einen effizienten Zugriff auf die Daten muß ein Modul direkt auf den Daten arbeiten können*.

  Da ein Modul unter Umständen sehr häufig auf die Daten zugreift (z.B. wird bei einem Marching Cubes Algorithmus das komplette Gitter durchlaufen und dabei auf dieselben Gitterpunkte mehrfach zugegriffen), müssen die Zugriffe direkt möglich sein.

- *Die Speicherung muß die volle semantische Information enthalten*.

Zunächst erscheint es ausreichend, daß zur Speicherung der Objekte eine Größeninformation gegeben wird, die es ermöglicht, dem Objekt einen entsprechend großen Speicherplatz zuzuweisen und beim Kopieren den entsprechenden Bereich, in dem das Objekt liegt, spezifizieren zu können. Wenn allerdings Daten zwischen Rechnern ausgetauscht werden müssen, die verschiedene Formate bei der internen Repräsentation von Fließkommazahlen benutzen, muß beim Verschicken der Datenobjekte von einem Request Broker zum anderen eine Konvertierung vorgenommen werden. Diese Konvertierung hängt natürlich davon ab, ob zum Beispiel eine Fließkommazahl oder eine Integerzahl verschickt werden soll.

- *Es muß möglich sein, ohne eine Änderung im System neue Datentypen einzuführen*.

  Um dem System die volle Flexibilität zu geben, auch an Situationen angepaßt werden zu können, die bei der Implementierung der Kernkomponenten noch nicht absehbar sind, ist es erforderlich, daß Module erstellt werden können, die eigene neue Datentypen benutzen. Die nach diesen neuen Datentypen erzeugten Datenobjekte muß der Request Broker ebenso handhaben können wie die nach bereits bekannten Datentypen erzeugten. Das schließt selbstverständlich auch nötige Konvertierungen ein.

Aus diesen Bedingungen ergibt sich sofort, daß eine umfassende Typisierung der zu speichernden Daten erforderlich ist. Dies bedeutet, daß jedes zu speichernde Element eine Typidentifikation mit sich bringen muß.

4.2.2 Hierarchie der Datentypen

Um die oben bereits angesprochene Flexibilität bei der Einführung neuer Datentypen zu gewährleisten, ist eine hierarchische Gliederung der Grunddatentypen eingeführt worden:

1. Auf der untersten Ebene befinden sich die einfachen Datentypen für Zahlen und Buchstaben. Eine solches Element wird gespeichert, indem zuerst eine Codierung des Datentypes und dann das eigentliche Datum selbst geschrieben werden. Die verfügbaren elementaren Datentypen sind `char`, `short`, `int`, `long`, `float` und `double`.
2. Der größte Teil der zu speichernden Daten wird in der Regel aus umfangreichen Feldern mit Elementen desselben Datentyps bestehen. Die zweite Hierarchieebene wird daher von Feldern gebildet. Beim Speichern eines solchen Feldes wird zunächst der Feldtyp geschrieben, dann die Anzahl der Elemente des Feldes und anschließend die Elemente selbst. Felder können aus den elementaren Datentypen bestehen.
3. Auf der dritten Ebene findet sich der Typ eines Zeigers. Dieser Zeiger verweist einfach nur auf eine andere Adresse innerhalb des Shared Data Space. Eine Adresse im Shared Data Space besteht aus einer Segmentnummer und einem Offset in dieses Segment. Die Segmentnummer ist nötig, um eine dynamische Erweiterung des Shared Data Space durch einfaches Hinzunehmen von neuen Segmenten zu ermöglichen, ohne alle bereits vorhandenen Daten umspeichern zu müssen. Ein Zeiger braucht als Speicherplatz für die Typcodierung, die Segmentnummer und den Offset. Analog zu den Feldern der zweiten Ebene gibt es auch ein Feld mit Zeigern.
4. Zur kompletten Erzeugung eines neuen Datentyps ähnlich einer Struktur in der Programmiersprache C dient die oberste Ebene. Mittels dieser Ebene lassen sich alle Elemente der unteren drei Hierarchiestufen zu einer neuen Gruppe zusammenfassen. Die prinzipielle Speicherung dieser Gruppe ähnelt der eines Character-Feldes: zunächst eine Typ-Identifizierung, dann die Länge des Feldes in Bytes, dann die einzelnen Elemente selbst. Im weiteren werden Datentypen der Ebene

vier als Strukturdatentypen bezeichnet.

Ein neudefinierter Datentyp wird über eine Typidentifizierung in das System aufgenommen. Zunächst gibt es die explizit festgelegten Grunddatentypen CHARSHM, SHORTSHM, INTSHM, LONGSHM, FLOATSHM, DOUBLESHM, sowie die Typen der zweiten Ebene CHARSHMARRAY, SHORTSHMARRAY, INTSHMARRAY, LONGSHMARRAY, FLOATSHMARRAY und DOUBLESHMARRAY. Die Zeiger der dritten Ebene sind an dem Typ SHMPTR und ihre Felder an SHMPTRARRAY erkennbar. Dann gibt es noch eine Reihe von Spezialtypen zur Verwaltung der Objekte (z.B. eine Objekt-Identifizierung). Ein zusammengesetzter Datentyp der Ebene vier ist ganz einfach daran zu erkennen, daß er zu keinem der anderen Datentypen gehört.

Wenn also beim Parsen des Shared Data Space eine Typidentifizierung auftritt, die keinem der fest eingebauten Datentypen entspricht, kann davon ausgegangen werden, daß es sich um einen neuen Datentyp handelt. Damit so ein neuer Datentyp vom Request Broker auch verarbeitet (d.h. verschickt und gegebenenfalls konvertiert werden kann), muß die gesamte Struktur aus den folgenden Datenelementen erkennbar sein.

Nach der Typnummer folgt genau wie bei einem Character-Feld die Länge des gesamten Objekts in Bytes. Daran schließt sich ein bei allen Datenobjekten von der Struktur her gleicher Header an, in dem alle Eigenschaften eines Datenobjektes stehen (Abbildung 4-5 zeigt ein solches Datenobjekt, nachdem es im Shared Data Space angelegt wurde und ein Anwendungsmodul das Handle Objekt für den direkten Zugriff erzeugt hat). Der Header enthält eine eindeutige Objektidentifizierungsnummer (ID, diese setzt sich aus dem Datum und der IP-Adresse des Rechners, auf dem das Objekt erzeugt wurde, zusammen), eine Versionsnummer (ver), ein Referenzzähler (refc), die Anzahl der Elemente, aus denen sich das Datenobjekt zusammensetzt (#el), ein Zeiger auf ein Character-Feld mit dem Namen (zgr) und ein Zeiger auf ein Feld mit einer Liste von Zeigern auf Character-Felder für die Speicherung von generellen Attributen (zgr). Die eigentlichen Daten folgen direkt im Anschluß; entsprechend der im Header gespeicherten Anzahl können nun beliebige Datenelemente in diese Struktur aufgenommen werden. Dies schließt selbstverständlich Zeiger ein, die wiederum auf andere neue Datenobjekte der vierten Ebene zeigen können. Dadurch ist es möglich, eine beliebige Datenstruktur in diesem Shared Data Space so zu repräsentieren, daß sie vom Request Broker bearbeitet werden kann.

Zu den bislang implementierten Strukturdatentypen gehören

- eine ganze Reihe von Gittertypen (rechtwinklige regelmäßige, rechtwinklig unregelmäßige, strukturierte, unstrukturierte, etc.) und dazupassende Datenstrukturen für die Speicherung von den Ergebnissen numerischer Simulationen.
- Punkte, Linien, Polygone und Volumen. Diese sind wiederum hierarchisch aufgebaut, d.h. in einem Volumendatenobjekt werden in einem Unterobjekt Polygone referenziert, deren Gesamtheit das Volumen bildet. Diese Polygone wiederum bestehen aus einem Linien-Unterobjekt und der zusätzlichen Information, wie diese sich zu Linien zusammenfinden. Und diese Linien bestehen aus Punkten und der Zusatzinformation, wie diese miteinander zu verbinden sind.
- eine Klasse von Strukturdatentypen, die nur andere Objekte enthalten, ohne zusätzliche Information hinzuzufügen. Die gebräuchlichsten sind die Menge (Set) und die Geometrie. Ein Objekt des Typs Menge ist nichts weiter als eine Klammer um eine beliebige Anzahl von Objekten beliebigen Typs; dabei ist die Anzahl beliebig und kann auch dynamisch verändert werden. Eine Geometrie besteht aus einem Objekt, das eine Basisgeometrie enthält (z.B. eine Polygonliste oder eine Menge von Linien) sowie aus zwei optionalen Zusatzobjekten, die dazu passend die Farbe und die Normalen enthalten.

- eine besondere Menge ist der sogenannte TimeSet. In dieser Menge befinden sich analog zu dem gewöhnlichen Mengenobjekt mehrere Elemente gleichen Typs, die aber zusätzlich in einem zeitlichen Zusammenhang stehen. Da für viele Module dieser zeitliche Zusammenhang unerheblich ist (beispielsweise bei der Berechnung einer Schnittfläche durch die einzelnen Zeitschritte), funktionieren die ganz normalen Funktionen einer Menge genauso auf einem TimeSet. Für Module, die auf die Zeitinformationen zugreifen müssen (wie zum Beispiel das Modul für eine Partikelverfolgung), gibt es entsprechend zusätzliche Schnittstellen.
- Multi-Media-Objekte, die darauf ausgelegt sind, Datenströme zu verarbeiten. Dies kann z.B. in der Form eines Ringpuffers erfolgen, d.h. es gibt bei einem Videodatenstrom Speicherplatz für eine gewisse Anzahl von Frames, die kontinuierlich dort abgespeichert werden. Ein Modul, das lesend darauf zugreift, kann ebenfalls permanent Frames auslesen.

Siehe hierzu auch [16] und [39].

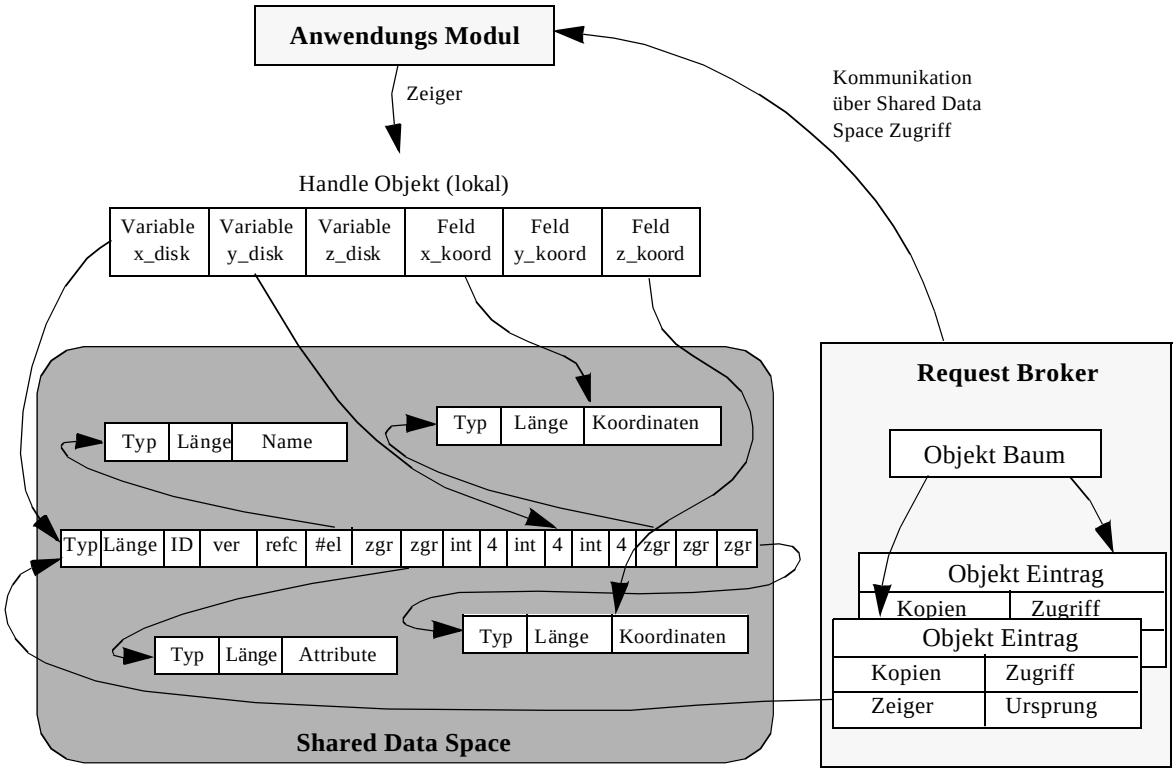

Abbildung 4-5: Zugriff auf den Shared Data Space

4.2.3 Zugriff eines Moduls auf die Datenobjekte

Erstellung eines Datenobjekts

Um ein neues Datenobjekt im Shared Data Space anzulegen, sind im wesentlichen nur der Platzbedarf und der Name, den das Datenobjekt tragen soll, nötig. Aus der Sicht eines Modulprogrammierers stellt sich der Ablauf sehr einfach dar: Er ruft einfach den Konstruktor für ein Objekt des gewünschten Datentyps auf und übergibt als Argumente den Namen und alle Parameter, die für die Größe des Objekts relevant sind (bei einem strukturierten Gitter sind das z.B. die Anzahl der Gitterpunkt in allen Richtun-

gen). Er erhält dann ein Objekt, bei dem er sich Zeiger direkt auf die zu beschreibenden Datenelemente holen kann (mehr dazu in Kapitel 4.6, Wesentliche Strategien der Implementierung).

Die Prozesse, die unterhalb dieser Programmierschnittstelle ablaufen, sind allerdings ein wenig komplexer. Aus der Information über das gewünschte Datenobjekt wird eine Message generiert, die an den Request Broker mit der Bitte um Erzeugung des Objekts und Bereitstellung des dafür benötigten Speicherplatzes geschickt wird. Dabei muß bereits der Datentyp der gewünschten Elemente mitgeschickt werden, da der Bedarf natürlich von den konkret gewünschten Datentypen abhängt. Der Request Broker überprüft daraufhin zunächst einmal, ob nicht schon ein Objekt des gewünschten Namens bei ihm oder einem anderen Request Broker existiert. Danach werden die einzelnen Anforderungen nach Speicherplatz bearbeitet. Dabei muß unterschieden werden, ob es sich um einfache Datenelemente handelt (einfache Zahlen oder Zeiger) oder um Elemente, die vom Objekt nur referenziert werden (wie z.B. Felder).

Basierend darauf berechnet der Request Broker den Speicherbedarf für die Speicherung des Objektes selbst (ein Objekt der Stufe vier aus 4.2.2) und stellt diesen bereit. Dann muß zunächst der Header gefüllt werden, d.h. es werden eine eindeutige Objekt-Identifizierungsnummer erzeugt, die Version und der Referenzzähler initialisiert, ein Null-Zeiger für ein Attributfeld und ein Zeiger auf den Namen des Objektes eingefügt (und der Bereich für das Character-Feld zur Speicherung des Namens reserviert und gefüllt) sowie die Anzahl der Datenelemente im eigentlichen Objekt eingetragen.

Die Datenelemente werden daran anschließend abhängig von ihrem Typ gespeichert. Datenelemente aus der Ebene eins werden direkt mit ihrer Typidentifizierung gespeichert. Handelt es sich um ein Feld aus Ebene zwei oder um einen Zeiger auf ein Objekt der Ebene vier, so wird nur der Platz für einen Zeiger reserviert. Bei einem Feld reserviert der Request Broker nun einen Speicherbereich der benötigten Größe und trägt dann dessen Adresse in den dafür reservierten Speicherplatz für einen Zeiger ein. Handelt es sich um einen Zeiger auf ein Objekt, so wird entweder der in diesem Fall gleich mitgelieferte Zeiger eingetragen oder der Zeiger als Null-Zeiger initialisiert.

Nachdem nun der benötigte Speicherbereich zugeteilt und die administrativen Daten des Datenobjektes initialisiert sind, schickt der Request Broker die Adresse (d.h. Segmentnummer und Offset) zurück an das Modul, das diese Anfrage stellt. Der gesamte Vorgang ist atomar, d.h. zwischen dem Eingang der Message mit der Bitte um Objekterzeugung und dem Zurücksenden der Adresse des erzeugten Datenobjektes führt der Request Broker keine anderen Services aus. Da dieser Vorgang in der Regel jedoch nicht allzu lange dauert, stellt dies keine Beeinträchtigung der Gesamtperformance dar. Probleme könnten höchstens auftreten, wenn die Zeiten, die für die Reservierung von Speicherplatz im Shared Data Space benötigt werden, aufgrund einer großen Anzahl von kleinen Objekten zu sehr wachsen. Dann ließen sich jedoch problemlos zwischen einzelnen Reservierungsvorgängen Abfragen einbauen, die nach den neuesten Messages fragen und diese gegebenenfalls sofort behandeln. Damit das Modul auf das neu erzeugte Datenobjekt zugreifen kann, sind mehrere Schritte notwendig, die in etwa denen entsprechen, die beim Zugriff eines Moduls auf ein bereits existierendes Objekt nötig sind. Diese werden im nächsten Abschnitt mitbehandelt.

Zugriff auf ein bereits existierendes Datenobjekt

Für den Fall, daß ein bereits existierendes Datenobjekt von einem Modul direkt zugegriffen werden soll, muß das Modul nur den Namen dieses Datenobjektes kennen. Ist zusätzlich zum Namen auch der Datentyp des Datenobjektes bekannt, ruft das Modul einfach den Konstruktor für diesen Datentyp auf und übergibt als Parameter den Namen. Dieser Name wird dann mittels einer Message an den Request Broker geschickt, der in seinem internen Objektbaum nach diesem Namen sucht. Findet er einen Eintrag,

so schickt er die Adresse des Datenobjektes an den Modul zurück. Der Kenntnisstand des Moduls ist jetzt der gleiche wie bei der Erzeugung eines neuen Objektes am Ende des vorigen Abschnittes: das Modul kennt den Datentyp und hat eine Adresse im Shared Data Space.

Als erstes wird jetzt die Typidentifikation an der ersten Stelle der Repräsentation des Datenobjektes im Shared Data Space mit der des aufrufenden Datentypkonstruktors verglichen. Stimmt der Typ nicht überein, wird der Zugriff mit einem Fehler abgebrochen. Ansonsten werden einige der Informationen aus dem Header in eine interne Datenstruktur kopiert. Da unter diesen Bedingungen die Anzahl und der Typ der Datenelemente des Objektes bekannt sind, lassen sich diese nun nacheinander für den Zugriff vorbereiten. Dabei können in einer solchen Elementliste nur Datentypen der Ebenen eins und drei vorkommen (Einzelelemente und Zeiger).

Die Vorbereitung sieht bei den verschiedenen Datenelementen folgendermaßen aus:

1. Bei den Datenelementen der Ebene eins wird ein Zeiger auf die tatsächliche Speicheradresse gelegt. Zugriffe auf dieses Datenelement werden durch Operator-Overloading wie eine normale Zuweisung in der Programmiersprache C gehandhabt (in beiden Richtungen; für Implementierungsdetails, siehe Abschnitt 4.6)
2. Ist das nächste Datenelement ein Zeiger, so wird dieser verfolgt und abhängig von dessen Typ weiterbearbeitet (auch hier wird die Zulässigkeit des Typs überprüft).
3. Felder der Ebene zwei werden ähnlich gehandhabt wie die Datenelemente der Ebene eins: auf den Beginn des eigentlichen Datenbereiches (vorher stehen der Typ und die Anzahl der Elemente) wird ein Zeiger gerichtet. Auch hier kann durch Operator-Overloading ein sehr natürlicher Zugriff auf die Feldelemente angeboten werden, der insbesondere auch überprüft, ob der Index auch im zulässigen Bereich liegt. Allerdings ist es für die Performance vorteilhaft, direkt unter Umgehung dieses Mechanismus auf das Feld zuzugreifen. Dazu kann sich der Modulprogrammierer einen Zeiger direkt auf die Feldelemente geben lassen. In Abbildung 4-5 ist dieser direkte Zugriff über den Feld-Eintrag zu erkennen, der direkt einen Zeiger auf die Daten enthält (für x_koord und z_koord dargestellt).
4. Trifft der Konstruktor beim Lesen auf einen Zeiger auf ein anderes Datenobjekt, so wird dieses ebenfalls sofort für den Zugriff vorbereitet. Dies ist eigentlich ein rekursiver Vorgang, d.h. man steht genau dort, wo man am Anfang des Zugriffes auf das gerade bearbeitete Objekt stand: man hat seine Adresse im Shared Data Space. Im Falle eines Polygons ist dann jedoch schon bekannt, daß es sich um ein Linien-Unterobjekt handeln muß. Bei einem Mengenobjekt ist jedoch nichts über den Typ bekannt.

Soll ein Datenobjekt gelesen werden, von dem der Typ noch nicht bekannt ist, so tritt ein Mechanismus in Aktion, den man als virtuellen Konstruktor bezeichnen könnte: man ruft ihn auf, ohne zu wissen, was als Ergebnis herauskommen wird. Da diese Möglichkeit von C++ jedoch nicht zur Verfügung gestellt wird, mußte ein entsprechender Mechanismus implementiert werden. Im wesentlichen wird basierend auf der im Shared Data Space vorhandenen Typ-Identifizierung aus einer Liste von vorher registrierten Datentypen der richtige herausgesucht und ein entsprechendes Objekt erzeugt. Der Modulprogrammierer hat dann die Möglichkeit, den tatsächlichen Typ abzufragen und basierend darauf in entsprechende Routinen zu verzweigen (genaueres hierzu findet sich in Kapitel 4.6, Wesentliche Strategien der Implementierung).

4.3 Persistenz

4.3.1 Definition von Persistenz

Persistenz aus dem lateinischen übersetzt bedeutet: „Bestehenbleiben eines Zustandes über längere Zeiträume“. Bezogen auf die Flüchtigkeit einer Information im Hauptspeicher eines Computers (wenn der Strom ausfällt, ist auch die Information verloren) und auch auf die durch die Laufzeit eines Programmes beschränkte Lebensdauer dieser Information, bedeutet Persistenz in diesem Zusammenhang:

„Persistente Objekte sind Objekte, die eine andauernde Existenz haben, d.h. eine Existenz, deren Dauer nicht durch das Ende einer Programmausführung begrenzt ist.“ [58]

Darüberhinaus ist Persistenz für die anderen Eigenschaften eines Datenobjektes nicht von Bedeutung:

„Persistente Objekte unterscheiden sich bezüglich anderer Eigenschaften wie z.B. Zustand, Verhalten oder Klassenzugehörigkeit nicht von nichtpersistenten Objekten.“ [53]

In einer Objekt-orientierten Umgebung läßt sich diese Unabhängigkeit besonders einfach dadurch deutlich machen, daß Persistenz eine Eigenschaft ist, die mit der Basisklasse, also der Klasse, von der alle Datenobjekte abgeleitet werden, verbunden ist. D.h. die Persistenzeigenschaft kann einem beliebigen Datenobjekt dadurch verliehen werden, daß man sie ihrer Basisklasse verleiht.

4.3.2 Auswirkungen der Persistenz

Bei herkömmlichen, dem Paradigma des visuellen Programmieren und des Datenflußnetzwerkes folgenden Applikationen bestehen zwischen den einzelnen Modulen Verbindungen, über die die Daten „fließen“. Selbst für den Fall, daß zwei Module auf demselben Rechner laufen und Shared Memory für den Datenaustausch zwischen ihnen genutzt wird, haben die Daten, obwohl sie im Shared Memory abgespeichert sind, zunächst keine eigene Identität. Für eine Performancesteigerung kann eine einfache Art von Persistenz in Form von Cacheing eingesetzt werden. Das heißt, die Daten werden nach Gebrauch nicht gleich wieder gelöscht, sondern bleiben im Shared Memory gespeichert und können so für den Fall, daß sie noch einmal gebraucht werden, direkt genutzt werden. Wenn eine solche Anwendung beendet wird oder die entsprechenden Module gelöscht werden, sind diese Daten jedoch verloren.

Der hier vorgestellte Ansatz behandelt diese Daten jedoch als eigenständige, nicht direkt mit der Lebensdauer der Module gekoppelte Objekte. Wenn ein Modul gelöscht wird und der dazugehörige Prozeß nicht mehr existiert, so hat dies zunächst einmal überhaupt keine Auswirkungen auf die von diesem Modul erzeugten Datenobjekte. Obwohl sich dieses Konzept zunächst nur auf die Laufzeit des Systems bezieht, stellt es doch eine logische Erweiterung des Datenflußnetzwerkes dar, Objekte nicht nur für die Laufzeit des Systems, sondern auch darüberhinaus zu erhalten.

Wichtig für die Handhabung von persistenten Objekten sind folgende Funktionalitäten:

- ein Objekt ist eindeutig identifizierbar
- ein Objekt läßt sich unabhängig von Modulen erhalten
- ein Objekt läßt sich über die Lebenszeit eines Programmlaufes hinweg erhalten, indem es auf ein permanentes Medium geschrieben wird (Festplatte, Band)
- es gibt eine Möglichkeit, diese Objekte hierarchisch zu strukturieren
- Objekte können frei während eines Programmlaufes auf den genutzten Rechnern verwendet werden

Identifizierung von Objekten

Anfangs wurde davon ausgegangen, daß innerhalb eines laufzeitorientierten Systems die Namengebung der Objekte angelehnt an den erzeugenden Modul ausreichend sei. Durch die Erweiterung auf persistente Objekte ist jedoch offensichtlich, daß so eine Kennzeichnung nicht mehr eindeutig ist. Daher werden alle Objekte jetzt mit einer Identifizierungsnummer versehen. Diese Objekt-Id wird aus der IP-Adresse des Rechners, auf dem es erzeugt wurde, der entsprechenden Zeit und der Prozeß-Id des erzeugenden Prozesses generiert. Diese Information dient alleine dem Zweck, eine eindeutige Identifikation zu generieren. Es wird nicht versucht, aus der Objekt-Id Rückschlüsse auf Erzeugungsort oder Zeit zu ziehen. Sollte dies nötig sein, müßten eigene Attribute, die diese Information enthalten, gesetzt werden.

Der Grund für diese Unterscheidung läßt sich an einem einfachen Beispiel ersehen, bei dem ein Objekt auf einen anderen Rechner kopiert wird: die Kopie eines Objektes auf einem anderen Rechner trägt denselben Namen wie das Originalobjekt. Das ist auch sinnvoll, denn das Objekt wird ja nur aus Gründen der Ablaufoptimierung kopiert; wenn die Kopie geändert wird, muß unbedingt auch das Original angepaßt werden. Es wird aber auch durchaus Zeiten geben, in denen Kopie und Original unterschiedlich sind, es also sinnvoll ist, daß diese beiden Objekte nicht dieselbe Objekt-Id haben. Dadurch würde aber auch der Versuch, aus der Objekt-Id die ursprüngliche Erzeugungszeit abzuleiten, zu einem falschen Ergebnis führen.

Objektlebensdauer unabhängig von Modulen

Während eines normalen Programmablaufes erlischt das Interesse des Benutzers an den Datenobjekten in der Regel in dem Moment, in dem er einen Modul löscht. Daher sind bei allen kommerziell erhältlichen Visualisierungsprogrammen, die auf dem Paradigma des visuellen Programmierens beruhen, in dem Moment, in dem ein Modul gelöscht wird, auch alle Daten, die er direkt produziert hat, verloren, es sei denn, der darauffolgende Modul hat sie noch in seinem Eingangsspeicher. Umgekehrt sind Daten, die sich nicht im Ausgangsspeicher des erzeugenden Moduls befinden, verloren, wenn der darauffolgende Modul gelöscht wird.

Durch die eigenständige Existenz der für den Datenaustausch zwischen Modulen verwendeten Datenobjekte ist nun jedoch gewährleistet, daß im Bedarfsfall diese Datenobjekte auch unabhängig von der Lebensdauer der Module weiter verfügbar sind. Dies wird offensichtlich, wenn man sich in Erinnerung ruft, daß diese Datenobjekte eigentlich nicht im Speicherbereich der Module liegen, sondern im Shared Data Space der vom Request Broker verwaltet wird. Zugriff erhalten die Module nur dadurch, daß sie den Shared Data Space in ihren eigenen Speicherbereich einblenden können. Entfernt werden können Datenobjekte allerdings nur durch den Request Broker; selbst der Controller kann nur eine Message mit der Bitte um Löschung an den Request Broker schicken.

Objektlebenszeit über das Ende des Programmlaufes hinaus

Ganz wesentlich für die Persistenz ist nun, daß der Request Broker vor Beendigung eines Programmablaufes die Möglichkeit hat, die Datenobjekte permanent abzuspeichern. Die einfachste Möglichkeit besteht darin, alle Datenobjekte vor Programmende in eine Datei zu schreiben und diese bei Programmbeginn wieder zu lesen. Diese Vorgehensweise erspart dem Benutzer im wesentlichen, die für seine Arbeit nötigen Daten jedesmal von neuem selbst aus einer Datei einzulesen.

Ein weitergehender und flexiblerer Ansatz würde den Shared Data Space als eine Art Cache betrachten. Das heißt, der Request Broker würde Objekte nur bei Bedarf einlesen und bei Speicherplatzmangel

automatisch die am längsten nicht benutzten Objekte auf das Permanentmedium auslagern. Dies würde gleichzeitig das gesamte Speichermanagement des Request Brokers wesentlich leistungsfähiger machen und einen effizienteren Einsatz knappen Hauptspeichers ermöglichen.

Hierarchische Strukturierung von Objekten

Durch die Persistenz ist es dem Benutzer ohne weiteres möglich, über einen gewissen Zeitraum hinweg eine große Anzahl von Objekten zu erzeugen. Wenn man den Idealfall betrachtet, daß diese Programmierumgebung die einzige vom Benutzer genutzte Anwendung ist, so werden alle vom Benutzer erzeugten und bearbeiteten Daten im Shared Data Space und seiner persistenten Erweiterung vorliegen. Um diese Datenmengen überschaubar handhaben zu können ist es unbedingt nötig, dem Benutzer eine Strukturierungsmöglichkeit zu geben. Das heißt, der Benutzer muß die Möglichkeit haben, seine Daten entsprechend seinen Vorstellungen hierarchisch gliedern zu können.

Als einfachste Möglichkeit bietet sich eine Dateisystemstruktur an, d.h. die Datenobjekte werden als einzelne Dateien unter ihrem Objektnamen abgespeichert. Parallel dazu wird eine Liste mit den Objekt-Ids gehalten. Zur hierarchischen Strukturierung dienen dann die normalen Verzeichnisstrukturen. Es besteht auch die Möglichkeit, Meta-Information in gesonderten Dateien abzulegen.

Unter Performance-Gesichtspunkten hat diese Lösung jedoch ihre Schwächen, da jeder Zugriff durch eine Reihe von Systemaufrufen und den Standarddateizugriff des Betriebssystems erfolgen muß. Es erscheint daher günstiger, einen exklusiven Bereich auf dem permanenten Medium zu reservieren und diesen unter eigener Regie möglichst optimal zu nutzen. Dies ermöglicht auch die einfache Nutzung von Möglichkeiten, die über die Standard-Dateisystemfunktionalitäten hinausgehen, wie zum Beispiel die Abspeicherung von Zugriffsinformationen oder die Nutzung mehrerer Links zwischen verschiedenen Datenobjekten.

Freie Nutzung verteilter Datenobjekte

Da es sich um ein grundsätzlich verteiltes System handelt, muß auch sichergestellt sein, daß auf Daten, die nicht auf dem lokalen Persistenz-Bereich gespeichert sind, zugegriffen werden kann. Das heißt, ein Zugriff auf Datenobjekte, die auf einem anderen Rechner persistent abgelegt sind, muß möglich sein. Durch die Verknüpfung der Request Broker ist dies jederzeit gewährleistet.

Um Überschneidungen bei der Nutzung der Shared Data Spaces in einer über mehrere Rechner verteilten Umgebung zu vermeiden, sollte unter Zuhilfenahme der im vorigen Absatz erwähnten Strukturierungsmöglichkeiten disjunkte Bereiche definiert werden. Zunächst sollte es einen vom jeweiligen Rechner unabhängigen Arbeitsbereich geben, in dem die für einen Programmablauf nötigen Zwischenobjekte abgelegt werden. Darüberhinaus steht auf jedem Rechner ein Bereich zur Verfügung, der die dort abgelegten persistenten Objekte beinhaltet. Diese persistenten Objekte lassen sich jederzeit auch in den Arbeitsbereich einbinden, wie sich auch temporäre Objekte durch Verknüpfen mit einem Persistenzbereich zu persistenten Objekten machen lassen. Für die einem Swap-Bereich vergleichbare Funktionalität läßt sich ein gesonderter Bereich nutzen, der nicht notwendigerweise die Persistenzeigenschaft aufweisen muß (aber durchaus kann, um Sitzungen einfach unterbrechen und später wieder fortsetzen zu können).

4.3.3 Vorteile der Persistenz

Die zeitlichen Größenordnungen, in denen die Arbeitsabläufe eines Ingenieurs stattfinden, der numerische Simulationen bearbeitet, reichen von in der Regel einigen Stunden bis zu mehreren Wochen. In Ausnahmefällen sind auch interaktive Berechnungen möglich, dies hängt jedoch stark von den verwendeten (Super-)Computern ab. Zur Zeit kann die Arbeitsweise zum Beispiel folgendermaßen aussehen:

1. der Nutzer erzeugt eine Geometrie, auf der die Berechnung durchgeführt werden soll
2. für diese Geometrie wird ein Gitter erzeugt
3. die Randbedingungen für die Berechnung werden bestimmt
4. die Berechnung wird durchgeführt
5. die Ergebnisse der Berechnung werden ausgewertet, d.h. visualisiert

Zwischen den einzelnen Schritten müssen Daten ausgetauscht werden, was in der Regel über Dateien erfolgt; d.h. aus dem Programm zur Geometrie-Erzeugung wird eine Datei mit den Geometriedaten geschrieben, die vom Programm für die Gittererzeugung eingelesen wird. Durch die Nutzung persistenter Objekte läßt sich diese etwas umständliche Schnittstelle deutlich einfacher handhaben: das Modul zur Geometrieerzeugung hat ein Ausgabedatenobjekt vom Typ Geometrie, das als Eingabeobjekt für die Gittergenerierung dient. Im Rahmen der visuellen Programmierumgebung werden einfach die entsprechenden Aus- und Eingänge der beteiligten Module verbunden und der Benutzer muß sich nicht um Dateien und deren Namen kümmern.

Beendet er eine Sitzung, so muß er nicht nacheinander alle Zwischenobjekte in Dateien abspeichern, vielmehr werden alle Datenobjekte, die nicht als nicht-persistent markiert sind, automatisch über den Persistenzmechanismus abgespeichert und stehen beim nächsten Programmaufruf sofort wieder zur Verfügung. Der Benutzer hat mittels der hierarchischen Strukturierung auch die Möglichkeit, Datengruppen aus verschiedenen Sitzungen jeweils getrennt zu halten und diese dann je nach Bedarf wieder anzukoppeln (von Speichern und Laden sollte in diesem Zusammenhang nicht gesprochen werden, da das Abspeichern kein explizit ausgelöster Vorgang ist, sondern automatisch vom System zur gegebenen Zeit durchgeführt wird; dies kann zum Beispiel auch dann passieren, wenn der zur Verfügung stehende Bereich im Shared Data Space knapp wird und Objekte, auf die länger nicht zugegriffen wurde, ausgelagert werden).

Für den Benutzer ergibt sich also eine Arbeitsweise, die deutlich näher an der realen Welt orientiert ist: wenn ein Handwerker ein Objekt bearbeiten will, nimmt er es in die Hand, wenn er seine Arbeit unterbricht, legt er es wieder weg. Am nächsten Tag ist das Objekt immer noch da, ohne daß explizit eine Handlung nötig ist, um klarzumachen, daß dieses Objekt noch gebraucht wird. Vielmehr ist es so, daß ein Objekt, das nicht mehr gebraucht wird, explizit weggeworfen wird.

4.4 Partitionierte Objekte

4.4.1 Definition der Partitionierung

Partitionierte Objekte unterscheiden sich von den anderen Objekten dadurch, daß sie aus mehreren Teilen bestehen, die unabhängig voneinander bearbeitet werden können (siehe auch [10]). Die Gesamtheit der Information ist jedoch nur dann vorhanden, wenn alle Teile (oder Partitionen) eines Objektes zur Verfügung stehen. Um ein sinnvolles Arbeiten auf einzelnen Partitionen zu ermöglichen, kann es nötig sein, Informationen mehrfach in verschiedenen Partitionen vorzuhalten. Ein einfaches Beispiel ist die Anwendung eines Isoflächen-Algorithmus auf ein partitioniertes Gitter (es ist unwesentlich, ob es sich

hierbei um ein strukturiertes oder unstrukturiertes Gitter handelt; der Einfachheit halber sei ein strukturiertes Rechteckgitter gegeben): damit auf einem Element (Hexaeder) des Gitters die dieses Element durchschneidende Isofläche berechnet werden kann, müssen sowohl die Koordinaten als auch die Daten auf allen Eckpunkten des Elementes bekannt sein. Sollte dieses Element auf dem Rand einer Partition liegen, so ergibt sich ganz zwangsläufig, daß auf dem angrenzenden Element der Nachbarpartition die Koordinaten und Daten auf den Randeckpunkten ebenfalls vorhanden sein müssen. Es ist also nicht möglich, ein solches Gitter einfach zu zerteilen bzw. zusammenzufassen; vielmehr müssen an den Partitionsgrenzen gewisse Redundanzen berücksichtigt werden.

4.4.2 Gründe für die Partitionierung

Es gibt im wesentlichen zwei Gründe dafür, den zusätzlichen Aufwand einer Partitionierung in Kauf zu nehmen: entweder die Datenobjekte sind so groß, daß sie nicht mehr an einem Stück verarbeitet werden können oder es läßt sich durch die Partitionierung eine Überlappung der Verarbeitung erreichen, die zu einer Verbesserung der Performance führt.

Datenobjektgröße

Insbesondere bei der Verwendung von massiv parallelen Rechnern können Datenmengen auftreten, die sich nicht mehr direkt auf einer normalen Workstation (wie sie bei einem Anwender normalerweise auf dem Schreibtisch steht) visualisieren lassen. Gerade bei einem massiv parallelen Rechner ist das Datenobjekt durch die Verteilung auf mehrere Prozessoren partitioniert, so daß sich eine Weiterverarbeitung dieses Datenobjektes als partitioniertes Datenobjekt anbietet.

Unabhängig davon, ob sich die Weiterverarbeitung noch auf dem Parallelrechner oder auf der Workstation des Benutzers abspielt, ist die Beibehaltung der Partitionierung die sinnvollste Methode. Findet die Weiterverarbeitung auf dem Parallelrechner statt, ist dadurch, daß die Daten auf dem Prozessor, auf dem sie auch erzeugt worden sind, bearbeitet werden, eine natürliche Verteilung der Arbeit gegeben. Dabei kann abhängig vom Algorithmus und den vom Benutzer gewählten Parametern eine nicht vorhersehbare Auslastung der Prozessoren erfolgen:

- bei der Bestimmung einer Isofläche in einem dreidimensionalen Gitter wird in der Regel das Gebiet komplett durchlaufen, da sich nicht von vorneherein bestimmte Partitionen ausschließen lassen.
- bei der Bestimmung der Schnittebene durch ein dreidimensionales regelmäßig partitioniertes strukturiertes Gitter läßt sich an Hand der Eckkoordinaten einer Partition direkt feststellen, ob die gesuchte Schnittfläche die Partition schneidet oder nicht.
- soll die Verfolgung einer Partikelbahn berechnet werden, so hängt die Nutzung der verschiedenen Partitionen dafür sowohl vom Startwert, den der Benutzer angibt, als auch von der numerischen Lösung ab. In der Regel ergibt sich hierbei eine absolut ungleichmäßige Auslastung der Prozessoren.

Findet die Weiterverarbeitung auf der Workstation statt, so ergeben sich ähnliche Konstellationen mit allerdings teilweise unterschiedlicher Bedeutung. Da die Workstation in der Regel über deutlich weniger Hauptspeicher verfügt, wird sie die Bearbeitung partitionierter Objekte stückweise, das heißt Partition für Partition, vornehmen. Für den Fall der Isofläche und der Schnittebene ergibt sich im wesentlichen der gleiche Ablauf wie bei der Weiterverarbeitung auf dem Parallelrechner: für jede Partition kann direkt die Bestimmung der Isoflächenteilstücke oder des Schnittebenenteils durchgeführt werden. Danach ist diese Partition erledigt und wird aus dem Speicher gelöscht, um Platz für die nächste Partition zu machen.

Im Fall der Partikelverfolgung ist dies jedoch nicht so einfach möglich, da sich erst aus der Berechnung der Partikelbahn ergibt, mit welcher Partition fortgefahren werden muß, das heißt, in welche Partition sich das Partikel bewegt hat. Dies kann dazu führen, daß sich bei einem im Kreis bewegenden Partikel dieselben Partitionen immer und immer wieder geladen und anschließend wieder gelöscht werden. Sollte der Speicherbereich auf der Workstation wirklich nur die Haltung einer Partition gestatten, muß jedesmal wieder neu vom Parallelrechner geladen werden; besteht die Möglichkeit, mehrere Partitionen entweder direkt im Shared Data Space oder wenigstens in einem per Cache beschleunigten Persistenzbereich zu halten, so läßt sich zumindest das dauernde Nachladen vermeiden (wobei es in einer High-Performance Netzwerk-Umgebung durchaus sein kann, daß eine HIPPI-Verbindung zum Parallelrechner schneller ist als der lokale Plattenzugriff auf der Benutzerworkstation).

Abhängig von der Größe der resultierenden Datenobjekte kann die partitionierte Bearbeitung nur für einen Teil der ganzen Visualisierung oder aber auch für den ganzen Prozeß genutzt werden. Wenn sie nur für einen Teil genutzt werden soll, so muß es einen Modul geben, der aus den partitionierten Objekten wieder ein ganzes Objekt erstellt, das dann von den weiteren Modulen standardmäßig weiterverarbeitet werden kann. Die Größe dieses Objektes muß allerdings für den benutzten Rechner geeignet sein. Dies kann insbesondere bei der Berechnung von Isoflächen problematisch sein, da diese abhängig vom benutzerdefinierten Isowert entweder sehr klein (bis hin zu einer „leeren“ Isofläche) oder aber in derselben Größenordnung wie der ursprüngliche Datensatz sind. Daher wird man in solchen Fällen von einer durchgehend partitionierten Verarbeitung ausgehen, das heißt, ein Modul reicht jede Partition, die er bearbeitet hat, sofort an den nächsten Modul weiter. Erst im letzten Modul einer Anwendung (in der Regel im Darstellungsmodul) werden dann die Partitionen zu einem Gesamtobjekt zusammengebaut; dies muß allerdings nicht ein Datenobjekt im Shared Data Space sein, sondern kann auch ganz einfach die interne Geometriepräsentation des Renderers sein.

Performance-Verbesserung

Die Partitionierung von Objekten kann sich auch ohne die Nutzung eines massiv parallelen Rechners als sinnvoll erweisen. Der wesentliche Vorteil besteht bei einem solchen Szenario darin, daß der Benutzer früher erste Darstellungen erhält.

Geht man von der Standardsituation aus, in der eine Datei eingelesen, die Daten bearbeitet und anschließend dargestellt werden sollen, so ergibt sich normalerweise folgendes Bild: die Datei wird zunächst komplett gelesen, dann werden gelesene Daten komplett gefiltert, als ganzes auf eine Geometriepräsentation abgebildet und anschließend dargestellt. Durch die Nutzung partitionierter Daten könnte sich folgender Ablauf ergeben: der Lesemodul liest den ersten Teil (z.B. ein hundertstel) der in der Datei gespeicherten Daten ein, schreibt sie in eine Partition und beginnt dann, den nächsten Teil zu lesen. Der darauffolgende Filtermodul beginnt daraufhin bereits mit der Arbeit und reicht die gefilterte Partition an den Geometriemodul weiter, der sofort die Partition für den letzten Schritt, die Darstellung erzeugt. Während der Lesemodul erst bei der dritten oder vierten Partition ist, sieht der Benutzer bereits die Darstellung der ersten Partition (dies funktioniert natürlich nicht so einfach, wenn man zum Beispiel eine Partikelverfolgung berechnet). Hinzukommt, daß normalerweise während eine Datei gelesen wird, die CPU nur zu einem Bruchteil ausgelastet wird, während bei einer partitionierten Verarbeitung diese ungenutzte CPU-Zeit bereits von den nachfolgenden Modulen genutzt wird. Es findet also zusätzlich zur subjektiven Beschleunigung auch eine objektive statt.

Noch deutlicher wird diese Beschleunigung wenn es sich um einen einfachen Mehrprozessorrechner handelt (zum Beispiel mit vier Prozessoren). Bei der nicht partitionierten Ausführung ist im wesentli-

chen immer nur eine CPU aktiv, während bei der partitionierten Ausführung bis zu vier Prozessoren gleichzeitig arbeiten. Dadurch läßt sich die Gesamtlaufzeit für diese Visualisierung deutlich reduzieren.

Ein ähnlicher Effekt ergibt sich bei der Verteilung von Modulen über mehrere Rechner: normalerweise wird erst ein Datenobjekt auf einem Rechner komplett angelegt und anschließend als Ganzes auf den Rechner auf dem der nächste Modul läuft übertragen. Während die beiden Request Broker damit beschäftigt sind, das Datenobjekt einzupacken, zu übertragen und wieder auszupacken, ist keines der anderen Module aktiv. Wird dagegen ein partitioniertes Objekt übertragen, können gleichzeitig sowohl vorhergehende Module mit nachfolgenden Partitionen beschäftigt sein und nachfolgende Module frühere Partitionen bearbeiten.

4.4.3 Die Synchronisierung des Zugriffes auf Partitionen

Um die sogenannte Pipelineverarbeitung der Partitionen der einzelnen Datenobjekte zu steuern, könnte zunächst der zentrale Controller genutzt werden. Das heißt, nachdem ein Modul eine Partition fertiggeschrieben hat, sagt es dem Controller Bescheid, der dann das nächste Modul darüber informiert, daß eine neue Partition zur Bearbeitung bereitliegt. Bei einer komplexeren Anwendung mit hochgradig partitionierten Datenobjekten können dadurch jedoch erhebliche zusätzliche Belastungen bei der Interprozeßkommunikation entstehen. Da es sich hierbei im wesentlichen um datenrelevante Vorgänge handelt, bietet es sich an, die Synchronisierung des Zugriffes auch über den Request Broker, der auch die sonstigen Datenzugriffe regelt, zu sichern.

Dies ist auch aus dem Grunde sinnvoll, daß Module, die ein neues Datenobjekt (auch wenn es sich nur um eine Partition handelt) bearbeiten möchten, dessen Adresse im Shared Data Space nur vom Request Broker erfahren können. Selbst wenn die Synchronisierung also über den Controller erfolgte, wäre doch bei jedem Schritt noch eine Kommunikation mit dem Request Broker nötig. Es liegt also nahe, die komplette Synchronisation direkt hierüber abzuwickeln.

Wenn ein Modul ein partitioniertes Objekt schreiben möchte, besorgt es sich vom Request Broker zunächst ein partitioniertes Datenobjekt. Dieses partitionierte Datenobjekt ist typunabhängig und beinhaltet alles, was für die Synchronisierung nötig ist. Die eigentlichen Partitionen sind ganz gewöhnliche Datenobjekte, die auch von den Kernalgorithmen der Module abgearbeitet werden. In der Regel kann man zwar davon ausgehen, daß bekannt ist, wieviele Partitionen vorhanden sind, allerdings gilt das nicht immer: zum Beispiel kann bei Isoflächen je nach Wert eine deutlich von der Partitionierung des Grundgebietes abweichende Partitionierung eintreten. Daher wird zunächst davon ausgegangen, daß es sich um eine noch nicht bekannte Zahl von Partitionen handelt. Der Einfachheit halber sollen auch die Nachbarschaftsbeziehungen zwischen den Partitionen zunächst nicht von Interesse sein (diese werden bei Partikelverfolgungen benötigt).

Nachdem der erste Modul ein partitioniertes Datenobjekt geöffnet hat, teilt er dies dem Controller mit und beginnt mit seiner Arbeit. Der Controller schickt daraufhin eine Startbotschaft zum nächsten Modul mit dem Hinweis, daß es sich um ein partitioniertes Objekt handelt. Dieser Modul erzeugt daraufhin wiederum ein partitioniertes Objekt für die Ausgabe von Datenobjekten, teilt dem Controller mit, daß er bereit ist und wartet darauf, daß die erste Partition seines Eingabeobjektes eintrifft. Der erste Modul hat in der Zwischenzeit seine erste Partition erzeugt und in das partitionierte Datenobjekt hineingesteckt.

Dadurch wird der Request Broker über das Vorhandensein einer neuen Partition informiert und unterrichtet wiederum alle Module, die sich für die Bearbeitung dieses partitionierten Datenobjektes angemeldet haben, über das Vorhandensein dieser Partition. Diese können jetzt auf die neue Partition im Shared

Data Space zugreifen und ihrerseits mit der Arbeit beginnen. Die Resultate dieser Arbeit werden in die partitionierten Ausgabedatenobjekte gesteckt, was die Benachrichtigung der darauf wartenden Module bewirkt. Nach Abschluß der Arbeiten warten die Module wieder auf eine neue Partition ihrer Inputobjekte.

4.4.4 Datentypen und Visualisierungsalgorithmen bei der Partitionierung

Die Partitionierung von Daten ist ein sehr aktuelles Problem, insbesondere im Hinblick auf die Nutzung von massiv parallelen Rechner, wie zum Beispiel der Cray T3E von SGI. Bei einem für die Parallelisierung geeigneten Algorithmus lassen sich die Rechenzeiten durch die Verteilung auf die verschiedenen Prozessor-Einheiten (PEs) zum Teil drastisch verkürzen. Gleichzeitig steht bei vergleichsweise normaler Ausstattung der einzelnen PEs mit Hauptspeicher (z.B: 128 MB pro PE) insgesamt dennoch ein immenser Gesamtspeicher zur Verfügung (bei 512 PEs über 60 Gigabyte). Bei einer numerischen Berechnung muß nun das Gitter, auf dem die Berechnung durchgeführt wird, auf die einzelnen PEs verteilt werden. Dabei ist wichtig, daß diese Verteilung gleichmäßig erfolgt, so daß die Berechnung möglichst schnell beendet werden kann (wenn alle Knoten bis auf einen in 10 Sekunden ihre Berechnung beenden können, dieser eine aber eine Stunde braucht, liegt die Gesamtberechnungsdauer bei einer Stunde). Ein weiterer Aspekt ist, daß die Grenzen zwischen den einzelnen Gebieten so klein wie möglich sein sollen, um den Datenaustausch zwischen den Gebieten so gering wie möglich zu halten.

Für die Partitionierung von Objekten in der Visualisierung können sich deutlich abweichende Anforderungen ergeben. Diese Anforderungen hängen wesentlich von der Visualisierungstechnik ab (Isofläche, Schnittfläche, Partikelverfolgung), aber auch von dem zugrundeliegenden Gitter (strukturiert: äquidistant, rechtwinklig, Multiblock; unstrukturiert: Tetraeder, Hexaeder, beliebig). In diesem Zusammenhang ist nicht der Vorgang der Zerlegung wichtig, sondern vielmehr die Auswirkungen, die das Bearbeiten partitionierter Objekte auf die verwendeten Visualisierungsalgorithmen hat. Im Idealfall funktionieren die Algorithmen auf den einzelnen Teilen eines partitionierten Objektes genauso wie auf einem einzelnen Datenobjekt.

Da dies in der Regel auch der Fall ist, sind die Strukturen zur Handhabung partitionierter Objekte entsprechend einfach ausgefallen: ein partitioniertes Objekt unterscheidet sich von einem nicht-partitionierten nur durch ein sogenanntes „Klammerobjekt“. Das heißt, es gibt, ähnlich zu einem „Set“-, also Mengenobjekt, einen Datentyp, zu dessen Rumpfdaten im wesentlichen (abgesehen von den Standarddatenobjektelementen) der Typ der in ihm steckenden partitionierten Objekte und deren Anzahl gehören. Dabei wird deutlich, daß ein partitioniertes Objekt homogen aus lauter Datenobjekten gleichen Typs besteht, die in ihrer Gesamtheit das ganze Datenobjekt ergeben. Diese Homogenität ist zwar nicht zwingend, vereinfacht jedoch die Handhabung außerordentlich.

Der Gittertyp

Welchen Einfluß hat nun der Gittertyp auf die Partitionierung? Wir gehen zunächst davon aus, daß die Partitionierung auf dem Datensatz, der visualisiert werden soll, bereits vorgegeben ist (was meistens der Fall sein dürfte). Bei einem strukturierten Gitter (d.h. gleiche Diskretisierung in jeder der Raumrichtung (nicht notwendigerweise gleichmäßig und orthogonal)) dürfte es sich in der Regel um eine quaderförmige Zerlegung handeln, d.h. einzelne Indexbereiche werden zu Blöcken zusammengefaßt. So führt

zum Beispiel bei einem 9 x 9 Gitter eine Zerlegung in 3 x 3 Blöcke zu einer Gesamtzahl von 16 Blöcken (siehe Abbildung 4-6).

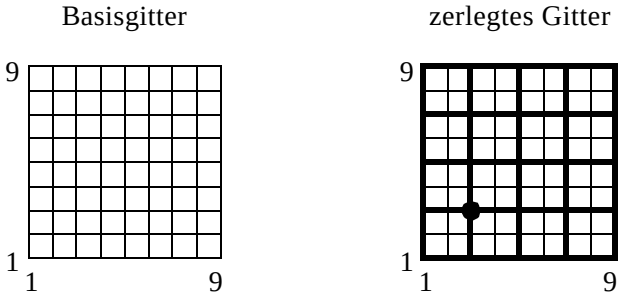

Abbildung 4-6: Partitionierung eines 9 x 9 strukturierten Gitters

Deutlich wird hierbei, daß die Zerlegung eines solchen Gitters zu redundanter Information führt. So gehört zum Beispiel der Punkt mit dem Index (2,2) (in Abbildung 4-6 mit einem Punkt markiert) gleich zu vier verschiedenen Partitionen. In diesem einfachen Beispiel wächst die Zahl der in diesem Gitter zu speichernden Gitterpunkte also von 9 * 9 = 81 auf 4 * 4 * (3 * 3)= 144, also nahezu auf das Doppelte. Im dreidimensionalen Raum wären im analogen Fall statt 9 * 9 * 9 = 729 sogar 4 * 4 * 4 * (3 * 3 * 3) = 1728, also deutlich mehr als das Doppelte an Punkten zu speichern. Daraus ergibt sich, daß bei zu kleinen Partitionen der Speicherbedarf überproportional groß ist.

Wenn man sich dagegen die durchzuführende Arbeit anschaut, entdeckt man, daß es z.B. bei einer Isolinienberechnung keinen zusätzlichen Aufwand gibt: im nicht-partitionierten Fall sind 8 * 8 = 64 Rechtecke zu untersuchen, im partitionierten 4 * 4 * 4 = 64. Wesentliche Unterschiede gibt es nur bei Algorithmen, die auf der Gitterstruktur beruhen, wie zum Beispiel die Berechnung einer Schnittfläche entlang der Gitterstruktur eines strukturierten Gitters (d.h. eine einem festgehaltenen Index entsprechende Fläche). Hier erhöht sich mit der Anzahl der Partitionen die Menge der zu überprüfenden Indexbereiche.

Diese Erkenntnisse lassen sich im wesentlichen (abgesehen von den indexabhängigen Algorithmen) auch auf unstrukturierte Gitter übertragen.

Für den Fall, daß die Partitionierung in der Visualisierung erst erzeugt werden muß, stellt sich das Szenario etwas anders dar. Dies dürfte hauptsächlich dann der Fall sein, wenn die Daten aus einer Datei eingelesen werden sollen und die Partitionierung genutzt werden soll, um mehrere Prozessoren mit Hilfe des Pipelining effizient zu nutzen. Im Fall eines strukturierten Gitters ist die Situation sehr einfach: bei linearer Speicherung werden so viele Daten der Indexebenen (bei dreidimensionalen Daten) gelesen, bis eine hinreichende Partitionsgröße erreicht ist. Die letzte Indexebene der vorhergehenden Partition wird dann auch als erste Indexebene der nächsten Partition genutzt und wiederum so lange Daten der Indexebenen gelesen, bis die Partition eine gewisse Größe erreicht hat. Es soll nicht unerwähnt bleiben, daß eine solche scheibenweise Partitionierung bei Visualisierungsalgorithmen wie zum Beispiel der Partikelverfolgung abhängig von den Daten sehr ineffizient sein kann. Anzustreben ist eine möglichst kugel- oder würfelförmige Form der Partitionen.

Sollte das Gitter jedoch unstrukturiert sein, läßt es sich nicht so einfach beim Einlesen partitionieren. Ein unstrukturiertes Gitter besteht in der Regel aus Daten über die Koordinaten der Eckpunkte und die Daten der einzelnen Elemente. Dazu können noch Nachbarschaftsinformationen kommen. Da die Zuord-

nung von Eckpunkten und Elementen normalerweise über die Elemente gesteuert wird (d.h. zu jedem Element ist gespeichert, welche Eckpunkte zu ihm gehören), die Numerierung der Eckpunkte jedoch völlig unabhängig von den Elementen sein kann, braucht man, um eine elementgesteuerte Partitionierung vorzunehmen, im Prinzip die volle Eckpunktinformation im Hauptspeicher. Wenn in der Datei jedoch zunächst die Element- und dann erst die Eckpunktinformation gespeichert ist, muß man den kompletten Datensatz einlesen, bevor man ihn partitionieren kann. Im umgekehrten Fall werden zunächst die Eckpunktdaten eingelesen und dann während des Einlesens der Elementinformation die Partitionen analog zum strukturierten Fall gefüllt. Auch hier ergeben sich die Partitionierungen ganz zwangsläufig aus der Reihenfolge, in der die einzelnen Elemente gespeichert sind, d.h. es können im Extremfall sehr ungünstige, möglicherweise sogar unzusammenhängende Partitionen entstehen, die wiederum bei der Partikelverfolgung zu großen Performance-Einbußen führen.

Der Visualisierungsalgorithmus

Bei den Visualisierungsalgorithmen kann man im wesentlichen unterscheiden zwischen denen, die

- Elemente jeweils nur lokal bearbeiten

und denen, die

- die Nachbarschaft der Elemente, die sie bearbeiten, mit einbeziehen.

Mit lokal soll in diesem Kontext alles, was zu einem Element oder einer Zelle eines strukturierten oder unstrukturierten Gitters gehört, bezeichnet werden.

Zu den ersteren gehören zum Beispiel die Schnittfläche (besser: Schnitt mit einer Hyperfläche), das Abbilden von Daten auf eine Geometrie (z.B. mittels Farbe oder Vektorpfeilen) oder das Croppen (Abschneiden von Teilen). Nicht-lokal sind zum Beispiel Partikelverfolgungen, topologische Untersuchungen und unter gewissen Bedingungen auch Isoflächenberechnungen.

Im Falle von lokalen Algorithmen lassen sich die Module für ein nicht partitioniertes Objekt praktisch ohne Änderungen im Berechnungsteil auch für partitionierte Objekte nutzen. Da die Nachbarschaft eines Elementes nicht von Bedeutung für das Ergebnis der Berechnungen für das aktuelle Element ist, läßt sich das gesamte Gebiet einfach zerteilen und partitioniert abarbeiten.

Ist der Algorithmus dagegen nicht lokal, so läßt sich das Konzept der Partitionierung entweder gar nicht oder meist mit deutlichen Performanceeinbußen anwenden. Dies wird besonders deutlich, wenn man die Partikelverfolgung betrachtet. Zunächst betrachten wir diesen Algorithmus im nicht partitionierten Fall: Der Benutzer wählt einen beliebigen Startpunkt für die Partikelbahn. Zuerst muß zu diesem Punkt das Element des Gitters gefunden werden, in dem er liegt. Falls es gefunden wird, wird basierend auf den Geschwindigkeitsdaten dieses Elementes die nächste Position des Partikels berechnet. Diese Position kann entweder im selben Element liegen oder aber in einem Nachbarelement (das nicht unbedingt direkt an das aktuelle Element anschließen muß).

Das Finden des Startpunktes im partitionierten Fall läßt sich noch relativ einfach durchführen: nacheinander werden alle Partitionen durchsucht, bis die, in der der Startpunkt liegt, gefunden wird (falls keine gefunden wird, hat der Benutzer offensichtlich einen Punkt außerhalb des Gebietes ausgewählt). Nachdem die Partition, die den Startpunkt enthält, gefunden ist, wird das Partikel analog zum nicht partitionierten Fall verfolgt. Dabei wird es früher oder später die Partition verlassen. In diesem Fall muß die Partition gefunden werden, in der das Partikel sich nun befindet. Das kann natürlich auch eine Partition sein, die bereits auf der Suche nach dem Startpunkt bearbeitet wurde. Im ungünstigsten Fall könnte sich

ein Partikel im Kreis immer wieder durch alle Partitionen hindurchbewegen, was zu einem permanenten Suchen und Neuladen von Partitionen führen würde. Der Aufwand ließe sich etwas reduzieren, wenn die Nachbarschaftsinformation zu den einzelnen Partitionen vorhanden wäre, man also nicht mehr suchen, sondern nur noch nachladen müßte.

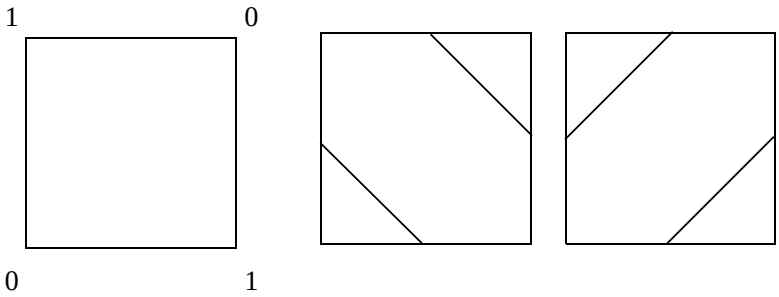

Abbildung 4-7: Uneindeutigkeiten bei Marching Cubes

Bei der Isoflächenberechnung ist nicht gleich zu erkennen, warum es sich unter Umständen um eine nichtlokale Aktion handelt. Dies liegt aber im meist benutzten Isoflächenalgorithmus, dem „Marching-Cubes“ [40] [80], begründet. Vereinfacht ausgedrückt wird bei diesem Algorithmus nach den Werten in den Eckpunkten geschaut und dann abhängig vom gegebenen Iso-Wert untersucht, ob eine Verbindungskante von der Isofläche geschnitten wird. Aus den Schnittpunkten wird dann ein Flächenstück gebildet, das mit allen anderen Flächenstücken die Isofläche bildet. Dies ist offensichtlich ein lokaler Vorgang, da kein Nachbarelement betroffen ist. Unter gewissen Umständen kann es jedoch zu nicht eindeutigen Konstellationen kommen. In Abbildung 4-7 ist eine solche Konstellation zu sehen. Dargestellt ist der Blick auf die Seitenfläche eines Hexaeders, der an die Nachbarpartition grenzt. Bei einem Isowert von 0,5 und den dargestellten Daten in den Eckpunkten ergeben sich zwei Möglichkeiten, die Isofläche durch die Randseite fortzusetzen. Hierin liegt auch die Nichtlokalität der Isoflächenberechnung bei partitionierten Objekten: das Nachbarelement in der nächsten Partition hat für die Fortsetzung der Isofläche dieselben beiden Alternativen zur Auswahl. Diese beiden Fortsetzungen müssen aufeinander abgestimmt werden, was im Fall des partitionierten Objektes zu einem Informationsaustausch mit dem Nachbarelement führt.

Darstellungen der Topologie eines Strömungsfeldes benötigen für ihre Berechnung zumindest die erste Ableitung, meist auch die zweite, der zugrundeliegenden Daten. Damit die sinnvolle Fortsetzung dieser Ableitungen über Partitionsgrenzen hinweg gegeben ist, müssen für ihre Berechnung auf Randelementen einer Partition die Daten auf den Randelementen der benachbarten Partition miteinbezogen werden.

Im Gegensatz zur Partikelverfolgung ist die Nichtlokalität der Isofläche und der topologischen Darstellungen allerdings begrenzt. Das heißt, daß von einer Nachbarpartition in der Regel nur die äußerste, abhängig von der benötigten Ableitung auch die zweitäußerste, Elementhülle benötigt wird. Diese läßt sich, wenn die Partitionierung von der Anwendung selbst durchgeführt wird, durch Überlappung jeweils lokal einführen. Wenn die Partitionen von einer anderen Anwendung übernommen werden, ließe sich noch ein Durchgang vorschalten, der diese Überlappung für die Partitionen erzeugt. Dies ist jedoch mit einem vergleichsweise hohen Aufwand verbunden.

4.5 High-Performance-Komponenten

Die Entwicklung der ganzen COVISE-Umgebung erfolgte unter den Vorgaben eines Arbeitsumfeldes, das von der Rechenleistung über I/O-Bandbreite und Vernetzung bis hin zur Graphikleistung von höchster Leistungsfähigkeit geprägt ist. Die Supercomputer, die am Rechenzentrum der Universität Stuttgart zur Verfügung standen, waren anfangs eine Cray 2 mit 4 Prozessoren und 2 Gigabyte Hauptspeicher, später eine Cray C94 mit 4 Prozessoren und 8GB und heute eine NEC SX4 mit 32 Prozessoren (a 2 GigaFLOPS) und 8 GB statischem (SSRAM) und 16 GB dynamischen (DRAM) Hauptspeicher sowie eine Cray T3E von SGI mit 512 Prozessoren und 64 GB Hauptspeicher. Als File-Server diente eine Cray M92 mit verschieden schnellen Dateisystemen (bis zu 50 MB/s) und Migrationsmöglichkeit auf einen Bandroboter. An Netzwerken standen (außer dem üblichen Ethernet) zur Verfügung: Ultranet, FDDI, HIPPI und ATM (bis zu 622 MBit/s). Als Graphikworkstations dienten mehrere Silicon Graphics Rechner bis hin zu einer Onyx2 Infinite Reality mit 14 Prozessoren und 4 GB Hauptspeicher.

Gemeinsam ist all diesen Komponenten, daß es in der Regel einiger Anstrengung bedarf, die optimale Leistung zu erreichen. Dabei ist es leider selbstverständlich, daß die von den Herstellern genannten Leistungsdaten nur in Ausnahmefällen erreicht werden können. Besonderes Anliegen bei der Entwicklung von COVISE war daher, dem Benutzer ohne zusätzlichen Aufwand soviel wie möglich von diesem Leistungspotential zugänglich zu machen. Dabei stellte sich des öfteren heraus, daß eine so effiziente Nutzung der Hardware, wie sie hier erreicht wurde, für einen „normalen“ Benutzer praktisch nicht möglich ist. Die Bestimmung der optimalen Parameter für die Netzwerke ließ sich oft nur über eine umfangreiche Reihe von Messungen unter den verschiedensten Bedingungen und Kombinationen erreichen (siehe dazu auch [21]). Mitunter waren (gerade bei den Netzwerken) auch besondere Einstellungen speziell für die Messungen nötig, die nur im Single-User-Mode durchgeführt werden konnten.

4.5.1 Supercomputer

Computer dieser obersten Leistungsklasse sind kompromißlos auf ein Ziel ausgerichtet: möglichst schnell zu rechnen. Diesem Ziel müssen sich alle anderen Aspekte unterordnen. Diese Ausrichtung hat natürlich Vor- und Nachteile. Wenn man ein Programm hat, daß für die spezielle Architektur und Konfiguration eines bestimmten Supercomputers geeignet ist, so wird dieses Programm auf diesem Rechner mit einer Geschwindigkeit ablaufen, die deutlich über der auf allen anderen Rechnern liegt. Auch die immensen Datenmengen, die bei einem Rechenlauf anfallen können, werden adäquat auf so einem Supercomputer verarbeitet (d.h. zum Beispiel, daß die Festplatten mit einer erheblich größeren Geschwindigkeit beschrieben und gelesen werden können).

Vektorisierung

Zu den Besonderheiten eines solchen Supercomputers gehören u.a. Vektorisierung, die Nutzung anderer Floating-Point-Formate oder auch das Fehlen solcher von Workstations her vertrauten Dinge wie Shared Memory.

Die Vorteile der Vektorregister bestehen darin, daß eine größere Anzahl von gleichen Befehlen, die auf verschiedenen zusammenhängenden Daten (in der Regel Felder) angewendet werden, sehr effizient ablaufen. Dazu wird eine gewisse Menge dieser Daten (abhängig von der Länge der Vektorregister) in diese Vektorregister geladen und anschließend dieser eine Befehl auf dem gesamten Vektorregister auf einmal ausgeführt. Meist werden diese verschiedenen Stufen (Laden, Dekodieren, Ausführen, etc.) überlappend ausgeführt; d.h. während ein Befehl auf einem Vektorregister ausgeführt wird, wird parallel schon der nächste Befehl vorbereitet. In der Regel handelt es sich bei diesen Befehlen um Additionen

oder Multiplikationen, d.h. die Vorteile der Vektorisierung kommen besonders bei numerischen Simulationen zur Geltung. Aber auch einfache Vorgänge wie das Kopieren von Feldern können mit Hilfe der Vektorisierung deutlich beschleunigt werden.

Datenkonvertierung

Ebenfalls von der Vektorisierung profitiert die Konvertierung von Daten. Die Cray Vektor-Supercomputer benutzen nicht das in der Workstationwelt vorherrschende IEEE Floating-Point-Format, sondern ein Cray-spezifisches. Um diese Daten auf der Workstation weiterverarbeiten zu können, ist es nötig, sie auf das IEEE-Format zu konvertieren. Dies wird am effizientesten von den von Cray angebotenen (vektorisierenden) Systemkonvertierungsroutinen bewerkstelligt. Da allerdings nur dann konvertiert werden soll, wenn dies auch wirklich nötig ist, wird beim Verbindungsaufbau geprüft, welches Zahlenformat die beiden Rechner benutzen und dann entschieden, ob und wenn ja, auf welchem Rechner konvertiert werden soll (siehe hierzu auch Kapitel 5.2, Messungen). So wird zum Beispiel bei einer Verbindung des Cray Compute Servers mit dem Cray File Server nicht konvertiert, bei einer gleichzeitig bestehenden Verbindung mit einer SGI-Workstation jedoch schon. Als Standard-Netzwerkformat wurde das IEEE-Format gewählt mit der Byteordnung der MIPS-Prozessoren. Sollte also ein Cray Vektorcomputer mit einem Intel-basierten Windows-NT PC kommunizieren, würde der Cray Rechner auf das IEEE Format konvertieren und der Intel-PC dann die Byte-Ordnung ändern.

Die Nutzung der Cray-eigenen vektorisierenden Konvertierungsfunktionen mag selbstverständlich erscheinen, ist es aber leider nicht: AVS zum Beispiel benutzt für die Konvertierung die xdr-Routinen. Dabei handelt es sich um zwar weitverbreitete, aber gerade für die Konvertierung von großen Datenmengen recht ineffiziente Routinen, die wohl der Einfachheit und der Portabilität halber zum Einsatz kommen, aber deutliche Geschwindigkeitseinbußen mit sich bringen.

Shared Memory

Eine weitere Besonderheit der Cray Vektor-Supercomputer ist das Fehlen von Shared Memory. Dadurch wurde es nötig, für die Kommunikation zwischen mehreren Prozessen, die auf einer Cray laufen, einen anderen Mechanismus zu finden. Da es keine der Effizienz von Shared Memory vergleichbare Möglichkeit gibt, wurden Modul-Code und Request Broker-Code in ein Executable integriert. Der Shared Data Space wird als lokaler Speicher (vom Heap) allokiert und kann von beiden Seiten des Codes zugegriffen werden. Gegenüber dem restlichen System erscheint dieses gemeinsame Executable wie ein kompletter Rechner mit einem Request Broker und genau einem Modul. Alle Verbindungen zum Controller und zu den anderen Request Brokern bestehen analog zu einer Shared-Memory-fähigen Umgebung (siehe Abbildung 4-8).

Die gesamten Abläufe der Kommunikation zwischen Controller und Modul, Modul und Request Broker und zwischen Request Brokern sind identisch zu denen auf einer Shared-Memory-Maschine. Ein Overhead, der sich nicht vermeiden läßt, ist die Replizierung von Datenobjekten auf demselben Rechner, wenn mehrere Module auf dem Supercomputer laufen. Dies hat den zusätzlichen Nachteil, daß gerade hier mit großen Datenmengen gearbeitet wird. Für die Zukunft ist jedoch auch in diesem Bereich Besserung in Sicht, da auch die leistungsstärksten Supercomputer zunehmend mit Shared-Memory-Funktionalität ausgestattet werden (z.B. die NEC SX4). Der Stellenwert, der dieser Funktionalität allerdings von den Herstellern beigemessen wird, wird darin deutlich, daß die NEC SX4 des Rechenzentrums bei 8 Gigabyte Hauptspeicher maximal 1 Megabyte Shared Memory pro Segment zuließ. Dieser Wert ließ sich durch Umkonfiguration des Kernels auf maximal 8 Megabyte erhöhen. Eine weitere Erhöhung auf

angemessene Werte ist nur durch eine Änderung im Betriebssystem möglich. Wenn also größere Datenmengen auf der NEC verarbeitet werden müssen, wird weiterhin auf den ineffizienten Ansatz der Integration von Modul und Request Broker ausgewichen werden müssen.

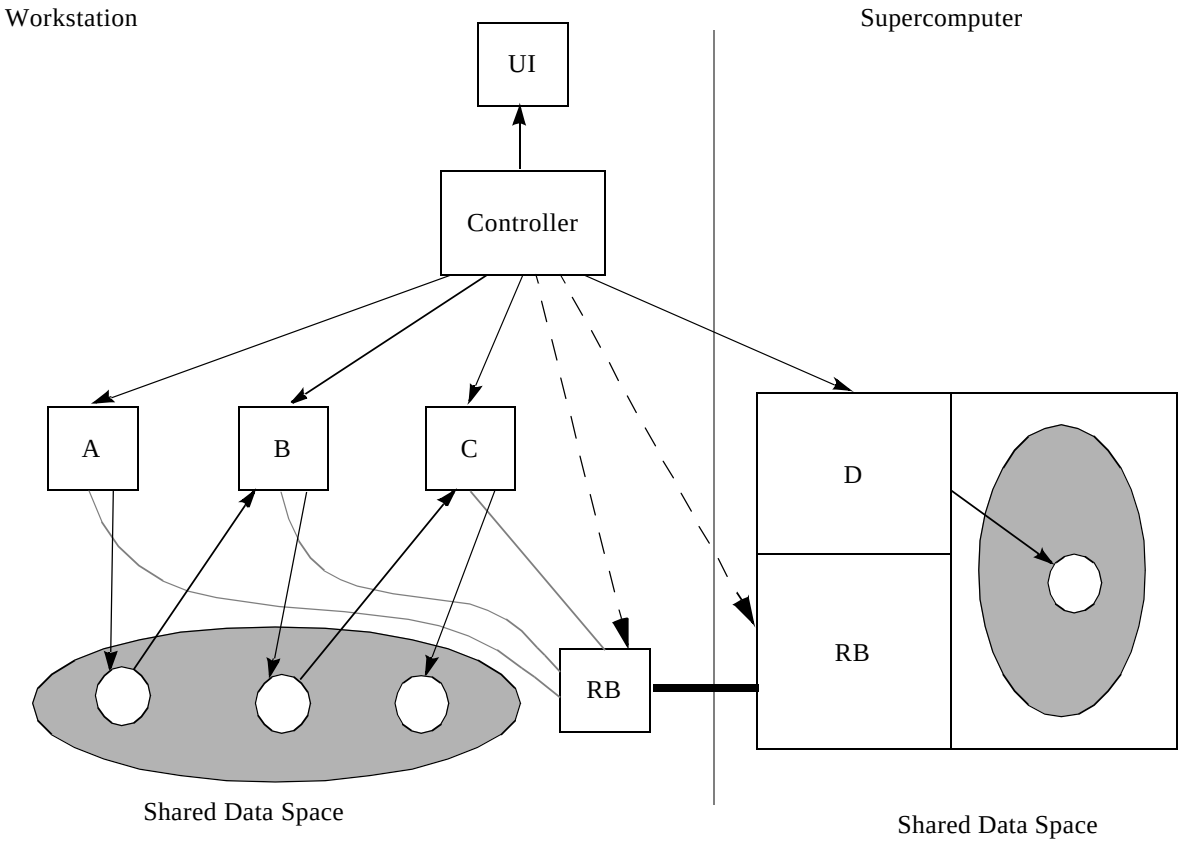

Abbildung 4-8: Modul-Request Broker auf einem Supercomputer

4.5.2 Netzwerke

Die verschiedenen heute verfügbaren Netzwerke haben sehr unterschiedliche Charakteristiken [21] [47] [66]. Als, wenn auch schon veraltetes, aber besonders deutliches Beispiel soll hier kurz das Ultranet dienen. Die theoretische Bandbreite dieses Netzwerkes beträgt 800 MBit/s. Dies wird unter anderem dadurch erreicht, daß der Overhead, der dadurch entsteht, daß jedes verschickte Datenpaket einiges an Information (z.B. die Zieladresse) mit sich tragen muß, minimiert wird. Die Pakete bei Ultranet sind daher extrem groß: 32 KB (zum Vergleich: die Ethernet-Paketgröße liegt bei 1.5 KB). Nun müssen nur noch genügend viele dieser Pakete pro Sekunde verschickt werden, um die 800 MBit/s zu erreichen.

Wenn eine Anwendung Daten über eine Ultranet-Verbindung verschicken möchte, tut sie das normalerweise mit den Standardparametern, die das Betriebssystem für jede neue Verbindung einsetzt. Ist die Standardpaketgröße zum Beispiel 4 KB, gehen bei jedem Datenpaket 28 KB leer über das Netz und die Leistungsfähigkeit dieser Verbindung ist von 800 auf 100 MByte/s reduziert. Selbst wenn alle anderen Voraussetzungen optimal sind, kann diese Übertragungsrate nicht überschritten werden. Wenn zusätzlich weitere Verluste in der Anwendung auftreten (z.B. werden abwechselnd große und kleine Datenmengen verschickt oder das Verschicken der einzelnen Pakete wird durch andere Berechnungen unterbrochen), bleibt für den Benutzer nur noch ein Bruchteil der maximal erreichbaren Leistung übrig,

ohne daß er daran etwas ändern könnte. Dieses Verhalten gilt, wenn auch in etwas anderer Form ebenso für HIPPI-Verbindungen. Bei diesen ist die Aufbauzeit relativ lang, der darauffolgende Datenaustausch dann jedoch relativ schnell. Auch hier werden somit die besten Ergebnisse bei maximaler Paketgröße erzielt. Bei ATM (das von Anfang an auf viele kleine Pakete ausgelegt wurde) ist dieses Problem jedoch nicht gegeben.

Ein weiterer Aspekt bei der Nutzung von Netzwerken ist, daß nur der Systemadministrator festlegen kann, auf welchem Wege ein anderer Rechner erreicht werden kann. Darüberhinaus ist mit dem Namen eines Systems oft nur ein Standardnetzwerkanschluß wie zum Beispiel Ethernet benannt, während die leistungsfähigeren Anschlüsse mittels eines Suffix an diesen Namen angesprochen werden müssen. Allein dadurch ist es dem Benutzer einer Standardanwendung in der Regel nicht mehr möglich, die schnelleren Verbindungen zu nutzen.

Diese Beispiele zeigen, daß zur optimalen Nutzung der verfügbaren Netzwerke Konfigurationsmöglichkeiten zur Verfügung stehen müssen, die es dem Benutzer erlauben, sich auch auf den Rechnern, auf denen er keinen direkten Einfluß auf die Konfiguration des System hat, die optimale Leistung verfügbar zu machen. Dazu genügt es auch nicht, einfach die gesamte Kommunikation über die schnellste Verbindung laufen zu lassen, wie insbesondere das Beispiel der Ultranet-Verbindung zeigt. Vielmehr muß es gezielt möglich sein, die Daten über eine andere Verbindung laufen zu lassen, als die Steuerungsinformation.

Die in COVISE realisierte Trennung von Daten-Verbindungen und Steuerungsverbindungen macht dies auf sehr einfache Weise möglich. Wenn man die Verteilung von Steuer- und Daten-Kommunikation auf die Verbindungen eines Szenario mit zwei Rechnern und gleichverteilten Modulen betrachtet, so wird nur ein geringer Bruchteil der Anzahl der Botschaften zwischen den beiden Rechnern über die Datenverbindung zwischen den Request Brokern ausgetauscht. Sieht man dagegen die Menge der ausgetauschten Daten, wird diese fast ausschließlich über die Verbindung zwischen den Request Brokern gehen. Es besteht daher die Möglichkeit, die Verbindung zwischen den Request Brokern über andere Interfaces gehen zu lassen, als die normalen Verbindungen zwischen dem Controller, den Modulen und dem Request Broker.

4.6 Wesentliche Strategien der Implementierung

4.6.1 Die Programmiersprache und verwendete Systemfunktionen

Die gesamte Software ist in der Programmiersprache C++ programmiert [67] [68]. Obwohl zu Anfang noch deutliche Schwierigkeiten bei der Verfügbarkeit bestanden (auf den Vektor-Supercomputern ist FORTRAN immer noch die Standardprogrammiersprache, gefolgt von C) und auch heute noch deutlich mehr Probleme bei der Verwendung von C++ im Vergleich zu C oder FORTRAN auftreten, überwiegen die Vorteile des objekt-orientierten Ansatzes die Nachteile bezüglich Verfügbarkeit, Fehlerhäufigkeit und Performance-Verlusten. Speziell im Hinblick auf mögliche Performance-Verluste wurde viel Wert auf eine effiziente Implementierung gelegt. So wurde zum Beispiel die oft recht teuere Technik der virtuellen Methoden (d.h. Methoden, die erst zur Laufzeit bestimmt werden, siehe dazu Abschnitt 4.6.2, Objekt-Orientierung) nur äußerst sparsam eingesetzt. Ebenso wurde generell auf sehr fortgeschrittene Sprachelemente wie zum Beispiel Exception-Handling oder Run-Time-Type-Identification (RTTI) verzichtet, um die Portabilität zu erhalten, obwohl diese auf den Workstation-Compilern verfügbar sind.

Bei der Benutzung von Systemfunktionen wurde ebenfalls darauf geachtet, die Portabilität nicht einzuschränken. Das heißt zur Kommunikation werden TCP/IP-Sockets, für den Shared Data Space wird

das System V Shared Memory verwendet. Dies sind Standardfunktionalitäten, die auf fast jedem UNIX-System verfügbar sind.

Shared Memory auf Cray Vektor-Supercomputern

Eine Ausnahme hiervon bilden nur die Cray Vektor-Supercomputer und das Fehlen von Shared Memory. Da die Benutzung von Shared Memory jedoch auf allen anderen Rechnern einen äußerst großen Performance-Gewinn mit sich bringt, wurde für den besonderen Fall der Cray Vektor-Supercomputer eine Möglichkeit gefunden, die Grundstruktur beizubehalten und die Änderungen am Quellcode so gering wie möglich zu halten.

Wie schon in Abschnitt 4.5.1, Supercomputer, erwähnt, wurden dazu der Anwendungsmodul und der Request Broker in ein einziges Executable kompiliert. Dadurch, daß die Schnittstelle zwischen diesen beiden Prozessen wohldefiniert ist (es gibt jeweils bei beiden Prozessen eine Routine zum Verschicken von Botschaften und eine Eingangsschleife, durch die alle eingehenden Botschaften kommen), war es nicht weiter schwer, sie zu kombinieren. Da Botschaften, die zwischen Prozessen verschickt werden, immer mit einer Typ-Identifizierung versehen sind, lassen sich auch Botschaften, die von außen kommen, eindeutig dem Modul- oder dem Request Broker-Teil zuordnen. Botschaften vom Modul an den Request Broker werden nun nicht mehr über einen Socket verschickt, sondern direkt als Argument an die Eingangsschleife des Request Brokers übergeben. Die Stellen, an denen bei Shared Memory Rechnern die Verwaltung dieser Shared Memory Segmente erfolgt, legen nun über den C++-Mechanismus `new` neue Speichersegmente an, die im weiteren Programmablauf genauso behandelt werden wie Shared Memory Segmente.

Die Kapselung dieser Differenzen ist so vollständig, daß der Quellcode für ein Modul, der auf einer Workstation, die über Shared Memory verfügt, der gleiche ist wie auf einem Vektor-Supercomputer ohne Shared Memory. Auf der Workstation laufen Modul und Request Broker jeweils als eigenständiger Prozeß; auf dem Vektor-Supercomputer ist der Request Broker zusammen mit dem Modul in ein Executable kompiliert. Aber auch der Aufwand für die Verwaltung des Bibliotheksquellcodes ist relativ gering, da sich alle Unterschiede nur an wenigen Stellen, innerhalb weniger Klassen befinden, die sich ohne weiteres über den `#define`-Mechanismus für die verschiedenen Rechner unterscheiden lassen.

4.6.2 Objekt-Orientierung

Die Nutzung einer objekt-orientierten Programmiersprache hat sich trotz des anfangs deutlich größeren Einarbeitungsaufwandes als ein sehr positiver Faktor erwiesen; dies gilt sowohl für die Wartung als auch die Weiterentwicklung der Software. Selbstverständlich läßt sich auch in C modular programmieren und auch eine gewisse Objekt-Orientiertheit erreichen (wenn auch nicht so elegant wie in C++). Die zwangsweise Beschäftigung mit Klassen, Objekten und Methoden in C++ macht es dem Programmierer jedoch wesentlich leichter, wirklich objekt-orientiert zu programmieren [54].

Wie bereits in Abschnitt 2.3 erläutert wurde, ist die Objekt-Orientierung im Hinblick auf die Datenobjekte des Shared Data Space nicht nach dem klassischen Objekt-Methode-Ansatz erfolgt, bei dem das Objekt im Mittelpunkt steht. Vielmehr wurde ein Ansatz verfolgt, bei dem Objekt und Methode (d.h. in diesem Fall Modul) gleichberechtigt nebeneinander stehen. Bei der Implementierung jedoch wurde die volle Bandbreite objektorientierten Programmierens genutzt. So wurde eine Prozeßklasse eingeführt, die die gesamte Kommunikation mit den anderen Prozessen sowie das Management des Shared Data Space übernimmt. Je nach den speziellen Anforderungen der einzelnen Prozeßtypen (Request Broker, Anwendungsmodul, Controller, etc.) werden einzelne Klassen abgeleitet, die die spezifischen Daten und

Methoden enthalten. Es gibt unter anderem Klassen für die Anbindung an den Shared Data Space, für Socket-Verbindungen, für das Ein- und Auspacken von Datenobjekten, die zwischen den Request Brokern verschickt werden, und für den Zugriff auf die Datenobjekte im Shared Data Space. In Abbildung 4-9 ist ein kleiner Ausschnitt aus der Klassenhierarchie für die Klassen, die den Anwendungsrahmen (d.h. die Kommunikation zwischen den Prozessen und den Zugriff auf den Shared Data Space) bereitstellen und die dem Modulprogrammierer den Zugriff auf die Verbindungen, über die die Botschaften ausgetauscht werden, erlauben, zu sehen. Diese Diagramme sind nach der Object Model Notation aufgebaut, der Notation der Object Modeling Technique (OMT) aus [54].

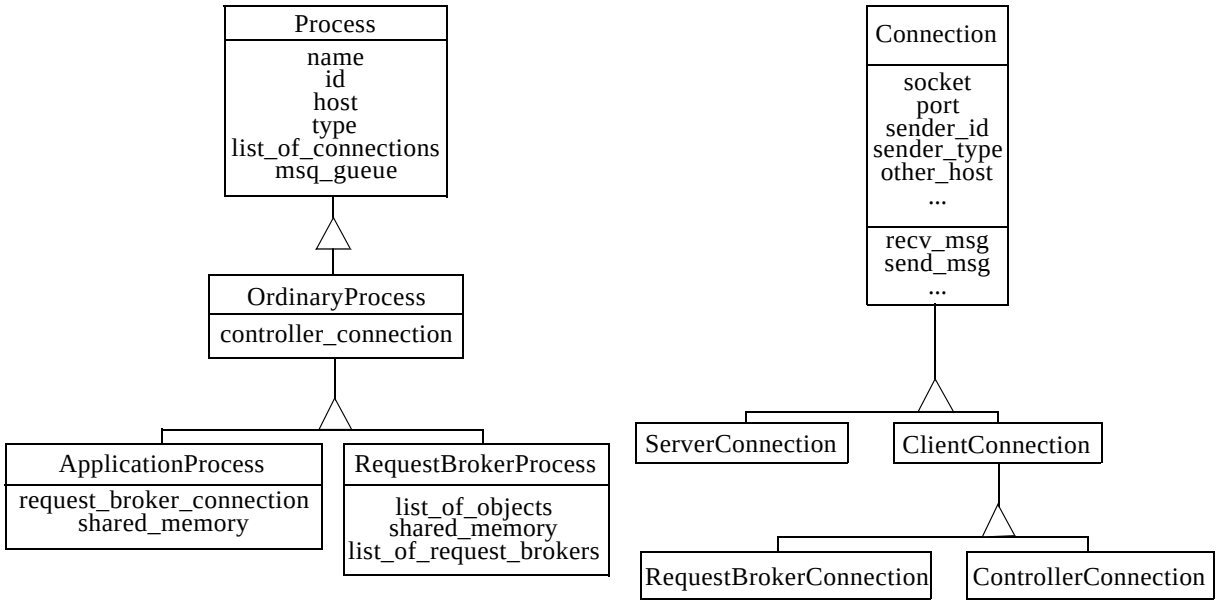

Abbildung 4-9: Ausschnitt aus der Klassenhierarchie

Der etwas detailliertere Aufbau des Request Brokers ist in Abbildung 4-10 zu sehen. Hier sind nur die Relationen der Klassen zueinander dargestellt, die Variablen und Methoden der einzelnen Klassen sind aus Gründen der Übersichtlichkeit weggelassen.

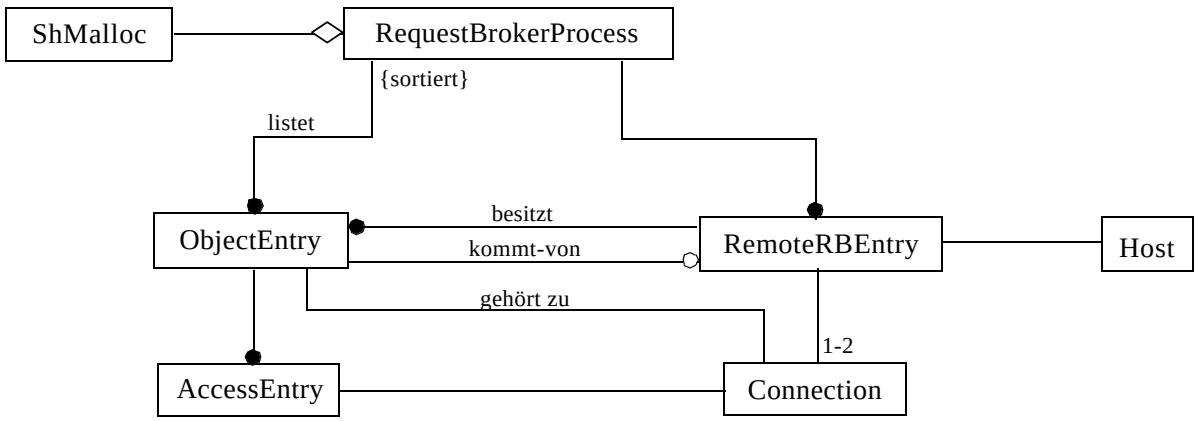

Abbildung 4-10: Die Klassen des Request Brokers

Ein RequestBrokerProcess hat einen SharedDataSpace (ShMalloc), keine oder viele ObjectEntries und keinen oder mehrere RemoteRBEntries (Informationen über andere RequestBroker). Ein ObjectEntry hat eine Liste mit den Zugriffen, die Verbindung zu dem Prozeß, der es erzeugt hat und, falls es sich um eine Kopie handelt, den RemoteRBEntry, von dem es herkommt. Ein RemoteRBEntry hat die Host-Information des Rechners, für den er die Informationen hält, die Verbindung zu diesem Prozeß (gegebenenfalls noch eine separate Verbindung für den optimierten Datentransfer) und eine Liste von ObjectEntries, die zu den Objekten gehören, die von diesem Host kopiert wurden.

4.6.3 Zugriff auf die Datenobjekte

In Kapitel 4.2 wurde bereits die Strukturierung der Datenobjekte im Shared Data Space beschrieben. Hauptgrund für die Art dieser Strukturierung ist, daß ein Request Broker in der Lage sein muß, ein Datenobjekt, das im Shared Data Space liegt, zu einem anderen Request Broker zu schicken, ohne daß er vorher die Struktur dieses Datenobjektes kennt; falls nötig, muß er sogar Datenkonvertierung vornehmen, also Information über die verwendeten Datentypen haben. Wenn ein Modul nun auf ein Datenobjekt zugreift, kann man zunächst einmal davon ausgehen, daß er die Struktur des Datenobjektes bereits kennt (sonst wüßte er ja nicht, was er damit anfangen soll). Dennoch kann die Strukturierung der Datenobjekte im Shared Data Space eine gute Hilfe dabei sein, festzustellen, ob auch wirklich ein konsistentes Datenobjekt im Speicher liegt. Siehe dazu auch [29] und [42].

In Abbildung 4-11 sind die bei einem Zugriff auf ein Datenobjekt im Shared Data Space aktiven Komponenten zu sehen. Mit Datenobjekt wird im weiteren eine Gruppe von Speicherstellen im Shared Data Space bezeichnet, in denen die eigentlichen Daten abgespeichert sind. Dazu zählen auch die Speicherstellen, auf die mit Zeigern aus diesem Objekt heraus verwiesen wird, obwohl diese aus Übersichtlichkeitsgründen in dieser Abbildung nicht mit einer Schraffur hinterlegt sind; diese Felder können auch weit entfernt vom Datenobjekt im Shared Data Space liegen (auch in einem anderen Segment). Als Datenelement wird ein Teil des Objektes, der einem Basisdatentyp wie Integer, float, Zeiger oder Feld entspricht, bezeichnet. Dazu gehören eine Typ-Identifizierung und das Datum selbst, also in dieser Abbildung (int, 4). Ein Zeiger-Datenelement enthält eine Typ-Identifizierung für den Zeigertyp, eine Shared Data Space Segment Nummer und den Offset in dieses Segment. Vor den eigentlichen Daten stehen eine Reihe von sogenannten Headerinformationen für dieses Datenobjekt; dazu gehören der Typ, die Anzahl der Datenelemente, die Versionsnummer, ein Referenzzähler, die Objekt-Id, ein Zeiger auf den Objektnamen und ein Zeiger auf die Attributliste. Dieser Header ist bei allen Datenobjekten im Shared Data Space gleich.

Das Zugriffsdatenobjekt

Für den Zugriff auf die Daten im Shared Data Space benutzt der Anwendungsmodul ein Zugriffsdatenobjekt. Dabei handelt es sich um ein ganz normales C++ Datenobjekt. Ein Teil dieses Objektes besteht aus den Standard C++-Datentypen und enthält zum Beispiel eine Kopie der Versions- und Typnummer des Datenobjektes. Der andere Teil des Zugriffsdatenobjekts besteht aus Zugriffselementen. Jedes dieser Zugriffselemente ist einem Datenelement im Shared Data Space zugeordnet und zeigt direkt auf die Speicherstelle, an der die für die Anwendung relevanten Daten stehen. Im Fall der Variablen `y_disk` zeigt der Zeiger direkt auf die 4, im Fall des Feldes `z_koord` direkt auf das erste Element des Datenfeldes im Shared Data Space.

Da jeder Datentyp aus einer Reihe von Basisdatentypen zusammengesetzt ist (siehe dazu die Aufzählung in Abschnitt 4.2.2, Hierarchie der Datentypen auf Seite 35), steht dem Modulprogrammierer für jeden dieser Basisdatentypen eine Klasse zur Verfügung, die den Zugriff auf ein Element dieses Typs im

Shared Data Space handhabt. Diese werden im weiteren Zugriffselemente genannt. Alle diese Zugriffselementklassen sind von der Basisklasse `ShmItem` abgeleitet, die alle Daten, die für einen Zugriff nötig sind, enthält:

```
class ShmItem {
protected:
     int shm_seq_no;// laufende Nummer des Shared Data Space Segments
     int offset;    // Offset in dieses Segment
     int type;      // Datentyp des Elements
     void *ptr;     // direkter Zeiger auf die Daten
public:
     ShmItem() { shm_seq_no = 0; offset = 0; type = NONE; };
     int get_shm_seq_no() { return shm_seq_no; };
     int get_offset() { return offset; };
     int get_type() { return type; };
     void *get_ptr() { return ptr; };
};
```

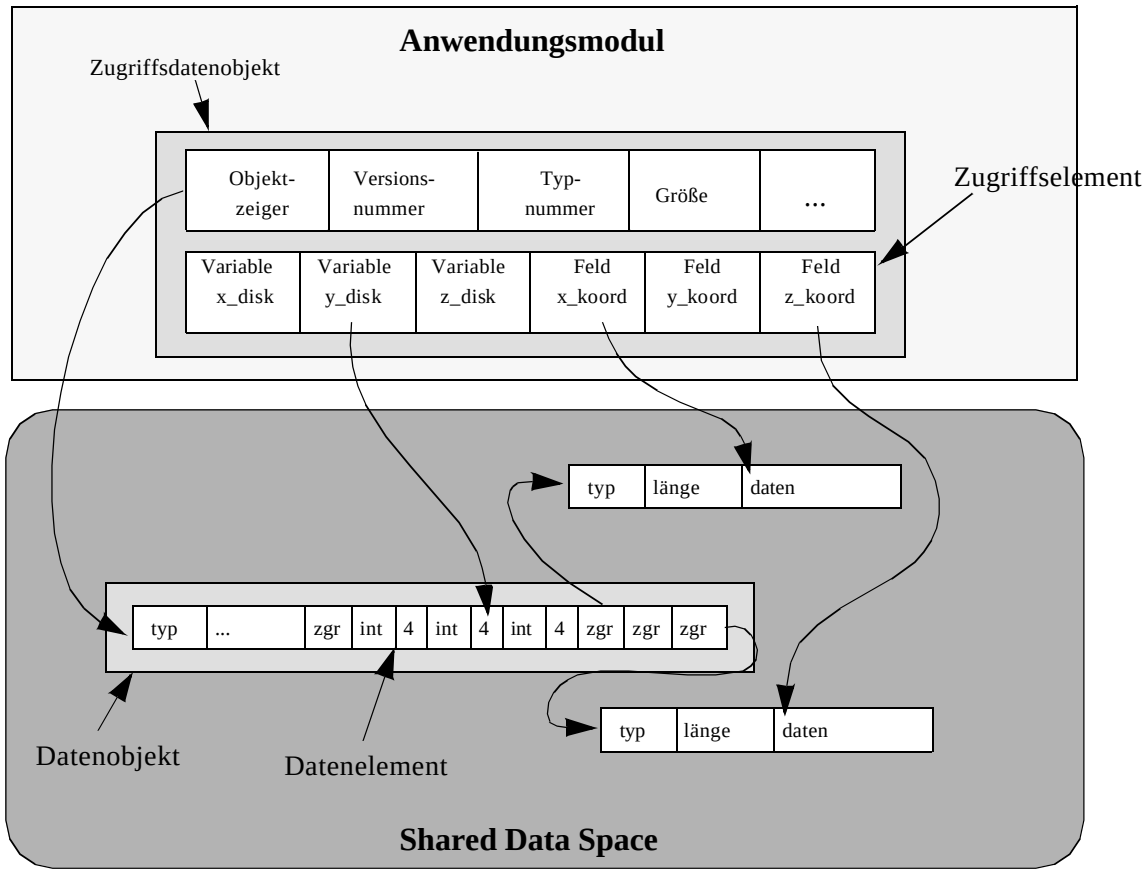

Abbildung 4-11: Zugriff auf den Shared Data Space

Von dieser Basisklasse sind die Klassen für den Zugriff auf einzelne Elemente oder Felder abgeleitet, die wiederum für die einzelnen Datentypen (character, short, integer, long, float und double) spezialisierte Unterklassen haben, sowie die Klassen zum Verwalten von Feldern von Zeigern und Feldern von Zeichenketten (siehe Abbildung 4-12). Direkt von der Klasse ShmItem sind die beiden Klassen ShmPtr und ShmArray abgeleitet, um Zeiger und Felder verwalten zu können. Diese beiden Klassen stellen im

Prinzip nur einen direkten Zugriff auf die eigentlichen Daten mittels der Methode `get_data_ptr` bereit. Bei der Feldklasse gibt es eine zusätzliche Variable, die die Länge des Feldes enthält.

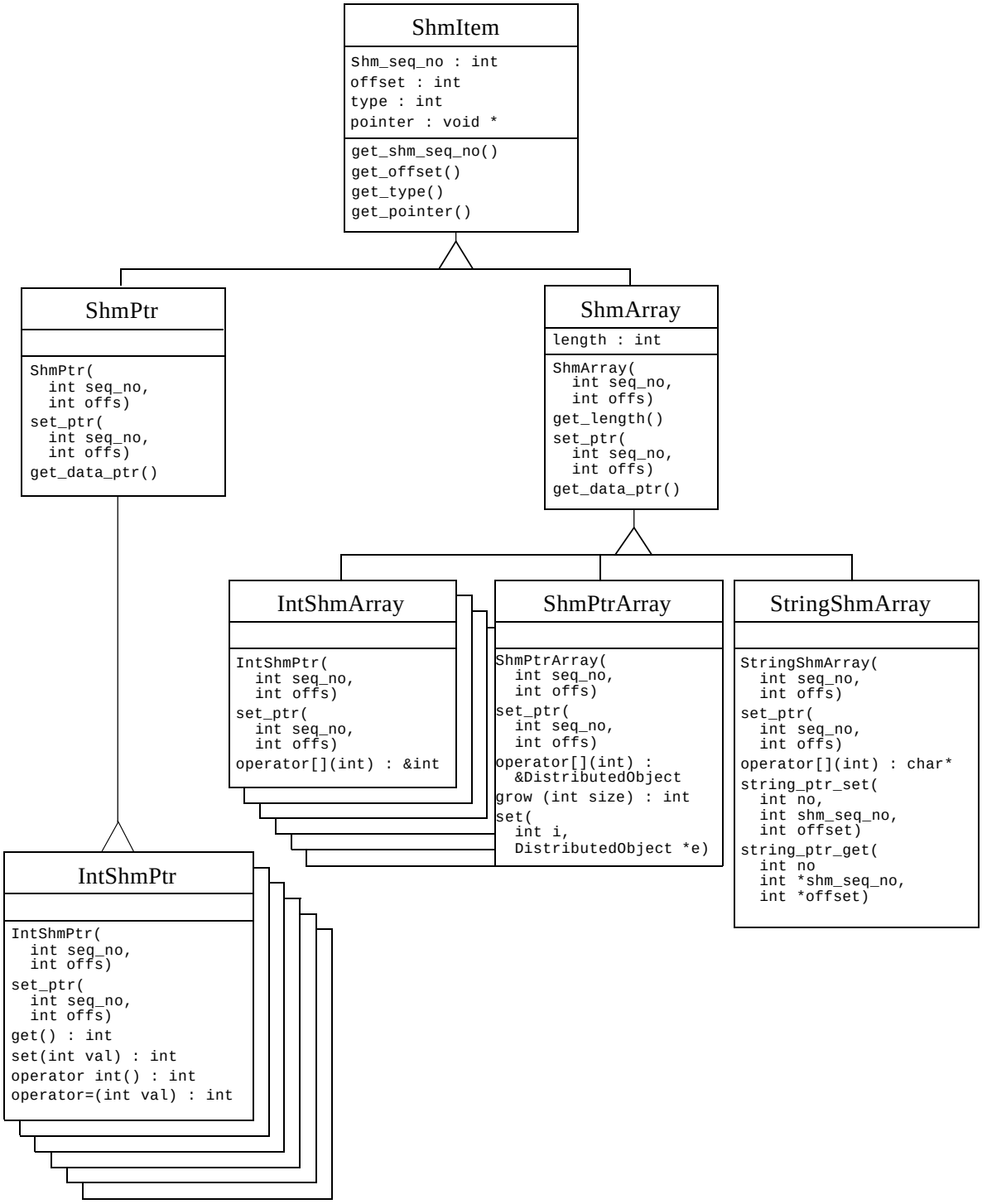

Abbildung 4-12: Klassenhierarchie der Zugriffselemente

Von diesen beiden Klassen sind jeweils für die oben genannten Datentypen entsprechende Unterklassen abgeleitet, die in ihren jeweiligen Initialisierungsroutinen Typkontrollen durchführen. Bei der Klasse

für Einzelelemente (z.B. IntShmPtr) kommen Methoden zum Setzen und Lesen dieser Variable hinzu, aber auch Operatoren für die Zuweisung und die Typkonversion. Diese Operatoren machen es den Modul- und besonders den Datentypprogrammierern sehr einfach, diese Variablen zu benutzen: sie können nämlich fast überall wie Variablen des C++ Grundtyps verwendet werden. Für die Feldklassen lassen sich noch eleganter die Klammer-Operatoren benutzen. Dadurch ist es sogar möglich, bei der Adressierung zu überprüfen, ob der Index innerhalb der Feldgrenzen liegt. Die Klasse `StringShmArray` dient der Speicherung von mehreren Character-Feldern und ist eingeführt worden, um das häufige Vorkommen von solchen Character-Strings einfacher handhaben zu können.

Die Klasse `ShmPtrArray` spielt eine besondere Rolle: im Unterschied zu den anderen Zugriffsklassen enthält sie nicht die Standarddatentypen, sondern zeigt wiederum auf andere Datenobjekte im Shared Data Space. Über diesen Mechanismus ist es möglich, eine Menge von Objekten (einen Set) effizient zu implementieren. Aus diesem Grund gibt es auch eine Methode `grow`, die es erlaubt, das Feld jederzeit zu vergrößern. Als Basisstruktur eines Sets ist es so möglich, beliebig viele Elemente zu addieren, ohne sich um die Feldgröße kümmern zu müssen.

Um den Zugriff auf die im Shared Data Space liegenden Daten eines Datenobjektes für den Modulprogrammierer so einfach wie möglich zu gestalten, wurde die Basisklasse `DistributedObject` eingeführt. In dieser Basisklasse wird die Grundfunktionalität sowohl für den Zugriff auf die Daten, als auch für die damit verbundene Kommunikation mit dem Request Broker bereitgestellt (wie zum Beispiel die Frage nach der Adresse eines bestimmten Datenobjektes im Shared Data Space oder die Rückgabe der Schreib- und Leserechte). Alles was mit der Struktur und den Datentypen eines konkreten Datentyps zu tun hat, wird in einer zu diesem Datentyp gehörenden, von `DistributedObject` abgeleiteten Klasse bereitgestellt. Diese Mechanismen sind so generell gehalten, daß es einem Modulprogrammierer ohne weiteres möglich ist, einen eigenen Datentyp zu definieren und die dazugehörende Datentypklasse von der Basisklasse abzuleiten. Die Objekte, die von dieser Basisklasse abgeleitet sind, werden im weiteren als Zugriffsdatenobjekte bezeichnet.

Spezielle Zugriffsdatenobjekttypen

Um das zu einem bestimmten Datenobjekt passende Zugriffsdatenobjekt von der Basisklasse `DistributedObject` abzuleiten, sind im wesentlichen zwei Schritte nötig: zuerst muß für jedes Datenelement des Objektes ein dazu passendes, von der Klasse `ShmItem` abgeleitetes Zugriffsdatenelement in die neue Klasse aufgenommen werden. Der Datenbereich der Klasse für ein strukturiertes nicht rechtwinkliges Gitter sieht folgendermaßen aus:

```
class DO_StructuredGrid : public DistributedObject{
     IntShm x_disc;         // Anzahl der Punkte in X-Richtung (X)
     IntShm y_disc;         // Anzahl der Punkte in Y-Richtung (Y)
     IntShm z_disc;         // Anzahl der Punkte in Z-Richtung (Z)
     FloatShmArray x_coord;    // X-Koordinaten (Länge X * Y * Z)
     FloatShmArray y_coord;    // Y-Koordinaten (Länge X * Y * Z)
     FloatShmArray z_coord;    // Z-Koordinaten (Länge X * Y * Z)
...
}
```

Der zweite Schritt besteht darin, auf für den Programmierer möglichst einfache Weise das Zugriffsdatenobjekt auf die dazugehörigen Elemente des Datenobjektes im Shared Data Space zu initialisieren. Dazu sind zwei Informationen wichtig: wo das Objekt im Shared Data Space liegt und aus welchen Datenelementen es besteht.

Die Lage des Datenobjektes im Shared Data Space erfährt das Zugriffsdatenobjekt über eine Anfrage beim Request Broker. Wenn das Datenobjekt bereits existiert, so muß nur der Name an den Request Broker geschickt werden. Dieser schickt dann die Adresse als Tupel (Nummer des Shared Data Space Segmentes, Offset in dieses Segment). Dazu gibt es in der Klasse `DO_StructuredGrid` den Konstruktor:

```
DO_StructuredGrid(char *name) : DistributedObject(name,"STRGRD");
```

Das zweite Argument im Aufruf des Konstruktors für die Basisklasse ist hierbei ein sechsbuchstabiges Kürzel für den Datentyp, das der Programmierer festlegt.

Existiert das Datenobjekt noch nicht im Shared Data Space, so muß außer dem Namen noch die Information über die Größe der Felder als Argument übergeben werden. Dazu dient der Konstruktor:

```
DO_StructuredGrid(char *name, int x, int y, int z) :
              DistributedObject(name,"STRGRD") ;
```

Weitere Konstruktoren können verwendet werden, um zum Beispiel weitere Initialisierungen vorzunehmen oder bereits existierende Gitterdaten gleich mitübergeben zu können und diese im Konstruktor sofort in den Shared Data Space zu schreiben. Zum Beispiel übergibt man diesem Konstruktor in den Variablen `xv`, `yv` und `zv` die Koordinaten für das Gitter; sie werden dann gleich in die entsprechenden Felder im Shared Data Space kopiert:

```
DO_StructuredGrid(char *name, int x, int y, int z, float *xv,
         float *yv, float *zv) : DistributedObject(name,"STRGRD") ;
```

Die Datenelemente, aus denen das Datenobjekt aufgebaut ist, muß der Programmierer in irgendeiner Form kodieren. Damit nicht für jeden Konstruktor dieselbe Liste kodiert werden muß und der Programmierer auch nicht für jede abgeleitete Klasse den Code zur Initialisierung der Zugriffselemente neu schreiben muß, wurde diese Funktionalität in die Basisklasse `DistributedObject` verlagert. Dazu dienen in der Basisklasse drei Funktionen:

```
int store_shared(int count ...);
int restore_shared(int count ...);
virtual int rebuild_from_shm();
```

Die beiden Funktionen `store_shared` und `restore_shared` sind einander recht ähnlich. Sie erwarten als Argumente zunächst eine Zahl, die die Anzahl der folgenden Argumente enthält. Mit diesen Argumenten teilt der Programmierer dem System mit, welche Datentypen die Klasse enthält und welche Adressen die Zugriffselemente haben.

store_shared

Die Methode `store_shared` wird benutzt, um ein neues Datenobjekt erstmals anzulegen. Im Fall der Klasse `DO_StructuredGrid` sieht der Aufruf von `store_shared` folgendermaßen aus:

```
store_shared(6, INTSHM, &x_disc, INTSHM, &y_disc, INTSHM, &z_disc,
          FLOATSHMARRAY, &x_coord, FLOATSHMARRAY, &y_coord,
          FLOATSHMARRAY, &z_coord);
```

Zur Beschreibung der Datenelemente eines Datenobjektes für ein strukturiertes Gitter braucht man sechs Argumente: drei Integer für die Feldgrößen und drei Felder von Floats für die Gitterkoordinaten. Innerhalb der Methode `store_shared` wird nun der benötigte Speicherplatz sowohl für das Datenobjekt selbst, inklusive seines Headers, als auch für die Felder bestimmt und mit der Bitte um die Erzeugung eines neuen Objekts und Zuweisung des dazugehörigen Speicherplatzes an den Request Broker

geschickt. Anhand der vom Request Broker zurückgelieferten Adresse werden dann die Zugriffselemente initialisiert. Die Methode `store_shared` wird immer in einem Konstruktor aufgerufen, im Falle des strukturierten Gitters zum Beispiel:

```
DO_StructuredGrid::DO_StructuredGrid(char *n, int x, int y, int z)
     : DistributedObject(n,"STRGRD") {
     x_coord.set_length(x*y*z);
     y_coord.set_length(x*y*z);
     z_coord.set_length(x*y*z);
     store_shared(6, INTSHM, &x_disc,
          INTSHM, &y_disc, INTSHM, &z_disc,
          FLOATSHMARRAY, &x_coord, FLOATSHMARRAY, &y_coord,
          FLOATSHMARRAY, &z_coord);
     x_disc = x;
     y_disc = y;
     z_disc = z;
}
```

Vor dem Aufruf von store_shared müssen die Längen aller Felder gesetzt sein, damit der entsprechende Speicherplatz beantragt werden kann. Nachdem der Speicherplatz zugewiesen und das Objekt im Shared Data Space erzeugt wurde, können die x-, y- und z-Variablen gesetzt werden.

restore_shared

In der Methode `restore_shared` wird basierend auf einem bereits existierenden Datenobjekt das Zugriffsdatenobjekt initialisiert. Zuerst wird dabei die Typinformation der Klasse, die `restore_shared` aufruft, mit der Typinformation, die das Datenobjekt im Shared Data Space in seinem Header trägt, verglichen. Stimmen diese beide nicht überein, wird die Initialisierung abgebrochen. Andernfalls werden nun die Headerinformationen des Datenobjektes zum Teil in das Zugriffsdatenobjekt übertragen oder die Zeiger des Zugriffsdatenobjektes initialisiert. Übertragen wird beispielsweise die Versionsnummer des Datenobjektes. Ändert sich diese im Shared Data Space. wird vor dem nächsten Zugriff eine Neuinitialisierung der Zugriffselemente durchgeführt. Die Zeiger für das Zugriffselement der Attribute wird initialisiert.

Wurde das Zugriffsdatenobjekt schon einmal initialisiert, wird die der Methode `restore_shared` sehr ähnliche Methode `update_shared` verwendet, um das Zugriffsdatenobjekt auf den neuesten Stand zu bringen. Normalerweise wird die Methode `restore_shared` aus einem Konstruktor heraus aufgerufen, befindet sich damit also im Einflußbereich des Programmierers des neuen Datentypes. Manchmal kann es jedoch auch nötig sein, diese Methode aus einem Teil des Systems aufzurufen, auf den der Modulprogrammierer keinen Zugriff hat. Dies ist zum Beispiel der Fall, wenn verschachtelte Datenobjekte angelegt oder rekonstruiert werden. Dabei wird die Methode `restore_shared` rekursiv auf dem geschachtelten Datenobjekt aufgerufen. Innerhalb der `restore_shared` Methode des anderen Datentyps ist zum Zeitpunkt des Kompilierens nicht unbedingt bekannt, um was für ein Datenobjekt es sich handelt, da dies auch von einer Benutzereingabe abhängen kann. Die Argumente, die für die Methode `restore_shared` des Datenobjekt in der Schachtelung gebraucht werden, sind daher nicht bekannt.

rebuild_from_shm

Um dieses Problem zu lösen, gibt es in der Basisklasse `DistributedObject` eine virtuelle Methode `rebuild_from_shm`. Für jede von `DistributedObject` abgeleitete Klasse muß der Pro-

grammierer eine Methode `rebuild_from_shm` zur Verfügung stellen. Diese Methode enthält im Prinzip nichts anderes als einen Aufruf der Methode `restore_shared` mit den für dieses Zugriffsdatenobjekt passenden Argumenten. Dadurch, daß diese Methode virtuell ist, wird sichergestellt, daß, auch wenn sie von einer Stelle aufgerufen wird, an der der eigentliche Datentyp, der zu diesem Datenobjekt gehört, unbekannt ist, die `restore_shared` Methode mit den richtigen Argumenten aufgerufen wird. Durch einen Aufruf der Methode `rebuild_from_shm` wird jederzeit die richtige `restore_shared` Methode aufgerufen, egal ob sie über einen Zeiger auf den richtigen Datentyp oder auf die Basisklasse `DistributedObject` aufgerufen wird.

In ähnlicher Weise können die Programmierer eines neuen Datentypes auch Informationen für die Darstellung eines Datenobjektes in der Benutzeroberfläche liefern. Dazu gibt es in der Basisklasse eine virtuelle Methode `get_class_object_info`. Als Argument wird eine Zeiger auf eine Liste von Zeichenketten geliefert, die dann abhängig vom konkreten Datentyp allokiert und mit den entsprechenden Informationen gefüllt wird.

Zugriff auf die Daten für den Modulprogrammierer

Obwohl die bislang beschriebenen Mechanismen für den Zugriff auf die Datenobjekte im Shared Data Space relativ komplex sind, gestaltet sich der Zugriff für den Modulprogrammierer sehr einfach. Dazu implementiert der Programmierer der Zugriffsobjektklassen eine Reihe von Methoden, die das Gewünschte liefern. Um an die Gittergrößen eines strukturierten Gitters zu gelangen muß der Modulprogrammierer nur die Methode `get_grid_size` aufrufen:

```
void get_grid_size(int *x, int *y, int *z) {
     *x = x_disc;
     *y = y_disc;
     *z = z_disc;
};
```

Bemerkenswert ist hierbei, daß es sich bei den Variablen `x_disc`, `y_disc` und `z_disc` nicht um einfache Integer handelt, sondern vielmehr um Objekte des Typs `IntShm`. Diese Klassen sind so implementiert, daß sie benutzt werden können wie ganz gewöhnliche Integer, das heißt sie können wie oben zugewiesen werden, aber auch auf der linken Seite einer solchen Zuweisung stehen. Analog dazu ist der Klammeroperator für die `ShmArray` Klassen überladen worden, so daß auch hier Zuweisungen in der gewohnten Weise erfolgen können. Zum schnellen Zugriff auf die Felder eines strukturierten Gitters gibt es noch die Methode `get_addresses`:

```
void get_addresses(float **x_c, float **y_c, float **z_c) {
     *x_c = (float *)x_coord.get_data_ptr();
     *y_c = (float *)y_coord.get_data_ptr();
     *z_c = (float *)z_coord.get_data_ptr();
};
```

Die Methode `get_data_ptr` gibt für eine beliebiges von der Klasse `ShmArray` abgeleitetes Feld den Zeiger auf das erste Datenelement im Shared Data Space zurück.

Eine für den direkten Zugriff gedachte Methode entbindet den Modulprogrammierer von jeglicher Index-Arithmetik:

```
get_point_coordinates(int i, int j, int k,
          float *x_c, float *y_c, float *z_c)
```

Zugriff auf die Datenobjekte im Modul

Wenn das Modul eine Startbotschaft vom Controller erhält, kommt für jeden Eingabeport der Name des an diesem Port liegenden Datenobjektes mit. Um auf dieses Datenobjekt zuzugreifen, ruft der Modulprogrammierer den Konstruktor dieses Datentypes mit dem Namen des Datenobjektes als Argument auf. Am Beispiel des strukturierten Gitters läßt sich dieser Zugriff sehr gut demonstrieren:

```
int x, y, z, i, j, k;
float xv, yv, zv;
DO_StructuredGrid *grid = new DO_StructuredGrid(„Gittername“);
grid->get_grid_size(&x, &y, &z);
for(i = 0;i < x;i++)
     for(j = 0;j < y;j++)
          for(k = 0;k < z;k++) {
          get_point_coordinates(i, j, k, &xv, &yv, &zv);
          compute(xv, yv, zv);
          }
```

Um seine Daten an ein Nachfolgemodul weiterzureichen, bedient sich der Modulprogrammierer eines mit einem Ausgabeport verbundenen Datenobjektes. Dieses Datenobjekt wird neu erzeugt, das heißt, der Modulprogrammierer muß beim Anlegen dieses Datenobjektes die Größen der Felder kennen. Um ein Datenobjekt vom Typ eines strukturierten Gitters anzulegen, müssen also im folgenden Beispiel `x`, `y` und `z` bekannt sein sowie die Felder `xv`, `yv` und `zv` die Koordinaten enthalten:

```
int x, y, z;
float *xv, *yv, *zv;
DO_StructuredGrid *grid;
grid = new DO_StructuredGrid(„neuer Gittername“, x, y, z, xv, yv, zv);
```

Sollten aus Gründen der Speichereffizienz die Daten direkt für den Shared Data Space erzeugt und in diesen geschrieben werden, kann dazu die Routine `get_addresses` genutzt werden.

Der virtuelle Konstruktor

Viele Anwendungsmodule lassen sich auf verschiedene Datentypen anwenden. So kann man zum Beispiel eine Schnittebene durch ein strukturiertes Gitter oder durch ein unstrukturiertes Gitter legen. Auch wenn das Ergebnis kaum zu unterscheiden sein mag, erfordert ein unstrukturiertes Gitter einen deutlich aufwendigeren Algorithmus als ein strukturiertes. Um jedoch nicht für jeden Gittertyp einen eigenen Anwendungsmodul bereithalten zu müssen (was auch die Übersichtlichkeit beeinträchtigen würde und den Benutzer nur mit einer unnötigen Komplexität verwirren würde) erlaubt man einem solchen Modul einfach mehrere Datentypen für den Eingang. Das Modul muß dann allerdings eine Möglichkeit haben, den Datentyp zu bestimmen und entsprechend in den geeigneten Algorithmus zu verzweigen.

Da mit der Startbotschaft vom Controller jedoch nur der Objektname übermittelt wird, muß das Modul erst auf das Objekt im Shared Data Space zugreifen, bevor er entscheiden kann, mit welchem Datentyp er es zu tun hat. Bei einem normalen Zugriff, bei dem das Modul weiß, um welchen Datentyp es sich handelt, ruft er einfach den Konstruktor der entsprechenden Klasse mit dem Namen des Datenobjektes auf (siehe oben). Weil er aber in diesem Fall noch nicht weiß, um was für ein Datenobjekt es sich handelt, muß es eine andere Möglichkeit geben.

Praktisch wäre es, wenn in diesem Fall das Prinzip der virtuellen Funktionen in C++ auch für den Konstruktor gelten würde: der Modulprogrammierer ruft einfach einen generellen Konstruktor auf, der im weiteren Verlauf feststellt, um welchen Datentyp es sich handelt und dann automatisch den Konstruktor dieses Datentypes aufruft und das passende Datenobjekt zurückliefert. Leider funktioniert dieses Prinzip des virtuellen Konstruktors nicht, es muß also eine andere Lösung gefunden werden.

Eine zwar insgesamt nicht ganz so einfache, aber für den normalen Modulprogrammierer (d.h. jemand, der nur mit den vorgefertigten Datentypen arbeitet) quasi identische Möglichkeit, wurde mittels einer statischen Liste der vorhandenen Datentypen geschaffen. Dazu muß sich jeder Datentyp, der dynamisch erkannt werden soll, zur Kompilierzeit in diese Liste eintragen. Dazu liefert er seine Typ-Identifizierung und die Adresse einer Funktion, die einen Konstruktor dieses Datentyps mit dem Namen des Objektes als Argument aufrufen kann und trägt dieses Paar in diese Liste ein. Trifft nun der Modulprogrammierer auf ein Datenobjekt, dessen Typ er nicht kennt, so übergibt er den Namen dieses Objektes an den Konstruktor der Basisklasse aller Datenobjekte. Diese wiederum schickt den Namen an den Request Broker, läßt sich von diesem dann die Adresse zurückschicken und nimmt eine Grundinitialisierung der datenstrukturunabhängigen Parameter wie Name oder Referenzzähler vor.

Möchte der Modulprogrammierer nun auf das eigentliche Datenobjekt zugreifen, so ruft er die zur Basisklasse gehörende Methode `create_unknown` auf. Diese Methode liest nun die Typ-Identifizierung aus dem Shared Data Space, besorgt sich aus der statischen Liste die Adresse des dazugehörigen Konstruktors und ruft diesen mit der Adresse des Objektes im Shared Data Space auf. Dieser Konstruktor initialisiert das Zugriffsdatenobjekt dann genauso, als ob er die Adresse direkt von Request Broker bekommen hätte. Als Rückgabewert der `create_unknown` Methode wird nun die Adresse des neuen Zugriffsdatenobjektes übergeben. Der Modulprogrammierer kann nun ganz einfach von diesem Zugriffsdatenobjekt den Typ abfragen und in die entsprechende Routine verzweigen.

Der Modulprogrammierer muß also zwei zusätzliche Aufrufe tätigen, bevor er ein Datenobjekt, dessen Typ er nicht kennt, bearbeiten kann. Ursprünglich war eigentlich geplant, diese beiden zusätzlichen Aufrufe in den Basiskonstruktor zu integrieren. Leider war das durch Inkompatibilitäten der Compiler nicht auf allen Plattformen möglich, so das dieser zwar etwas umständlichere, aber dafür portable Weg gewählt wurde. Der Programmierer eines neuen Datentypes muß darüberhinaus noch die Methode, die den Konstruktor aufruft, schreiben und seinen Datentyp in der statischen Liste quasi registrieren.

Kapitel 5

Resultate

Wie bereits in der Einleitung erwähnt, ist die in dieser Arbeit vorgestellte Datenverwaltung Teil eines verteilten und kollaborativen Visualisierungssystemes: COVISE. Das Ausgangsprojekt PAGEIN (Pilot Applications in a Gigabit European Integrated Network) hatte zum Ziel, kollaboratives Arbeiten in der Predesign-Phase der Entwicklung eines Flugzeugs einzuführen und die dafür nötigen Werkzeuge bereitzustellen. Die Projektpartner Daimler Benz Aerospace Airbus, Aerospatiale, British Aerospace und Defense Research Agency lieferten dafür das Anwenderszenario, DLR, NLR, ONERA und CIRA waren zusammen mit dem RUS für die Durchführung verantwortlich, genauer gesagt für die Entwicklung der nötigen Werkzeuge [49] [50].

Das Szenario des PAGEIN-Projektes bestand aus der kollaborativen Analyse einer Simulation, die auf einem Supercomputer läuft. Das heißt, es waren zunächst einmal die Grundzüge eines verteilten Systems zu analysieren und ein Rahmenwerk zur Verfügung zu stellen, in dem die gestellte Aufgabe angegangen werden konnte. Eine Analyse der Bedürfnisse und der bereits existierenden Software auf diesem Gebiet führte zu dem in dieser Arbeit vorgestellten Design. An dieser Stelle soll nicht erneut auf die Effizienzanforderungen, die an ein solches Werkzeug gestellt werden, eingegangen werden; die bei der Entwicklung eines Flugzeuges in der Regel anfallenden Datenmengen erfordern zweifellos einen äußerst effizienten Umgang mit Ressourcen aller Art.

Im Folgeprojekt ADONNIS (A Demonstration Of New Network Interoperable Services, ESPRIT, Nr. 9033) wurde mit Hilfe weiterer Partner (u.a. Dassault und CASA) die gesamte Software um zusätzliche Funktionalität erweitert und an neue Anwendungsgebiete (in diesem Fall die verteilte Entwicklung von Satelliten) angepaßt. Die Zahl der auf COVISE (und damit auch auf dem Datenverwaltungssystem dieser Arbeit) aufsetzenden oder es weiterentwickelnden Projekte liegt inzwischen über zehn. Dazu zählen mehrere Projekte, die eine Anpassung an die Bedürfnisse der Automobilindustrie zum Inhalt haben (u.a. die Visualisierung von numerischen Simulationen der Vorgänge im Zylinder eines Verbrennungsmotors oder von Crash-Berechnungen).

Aus der Vielzahl der verfügbaren Anwendungen sollen drei exemplarisch herausgegriffen werden, um an ihnen die Vorteile des Systems zu verdeutlichen.

5.1 Anwendungsbeispiele

5.1.1 PEGASE: Berechnung einer Wirbelauflösung

Der Numerikcode PEGASE (von ONERA im Rahmen des PAGEIN-Projektes zur Verfügung gestellt) berechnet auf einem nicht-äquidistanten regulären Hexaedergitter den Vorgang einer Wirbelauflösung. Als Anfangsbedingung wird auf diesem Gitter ein Wirbel vorgeschrieben. Zur guten Auflösung des zylindrischen Wirbels in der Mitte der beiden radialen Koordinatenachsen ist das Gitter dort feiner

aufgelöst als am Rand. Mit fortschreitender Berechnung löst dieser Wirbel sich immer mehr auf, das heißt seine anfangs klare Struktur franst aus. Zur Visualisierung eines solchen Resultates eignet sich am besten eine Isofläche, die konstant mit dem Betrag der Geschwindigkeit ist. Auf diese Isoflächen ließen sich dann andere Werte wie zum Beispiel die Temperatur oder die Wirbelstärke abbilden (mit Hilfe eines sogenannten Colormap-Moduls).

Die bei der numerischen Berechnung anfallenden Datenmengen können sehr groß werden. Die Daten, die zur Darstellung der Isofläche benötigt werden, sind vom Umfang her in der Regel erheblich geringer, so daß es sich anbietet, die Berechnung der Isofläche noch auf dem Supercomputer, auf dem auch die numerische Rechnung läuft, vorzunehmen. Dies gilt ebenso für die Abbildung der anderen Daten auf die Isofläche. Für den Datenfluß ergibt sich das in Abbildung 5-1 dargestellte Bild. Aus Gründen der Übersichtlichkeit wurden nur die wesentlichen Komponenten eingezeichnet.

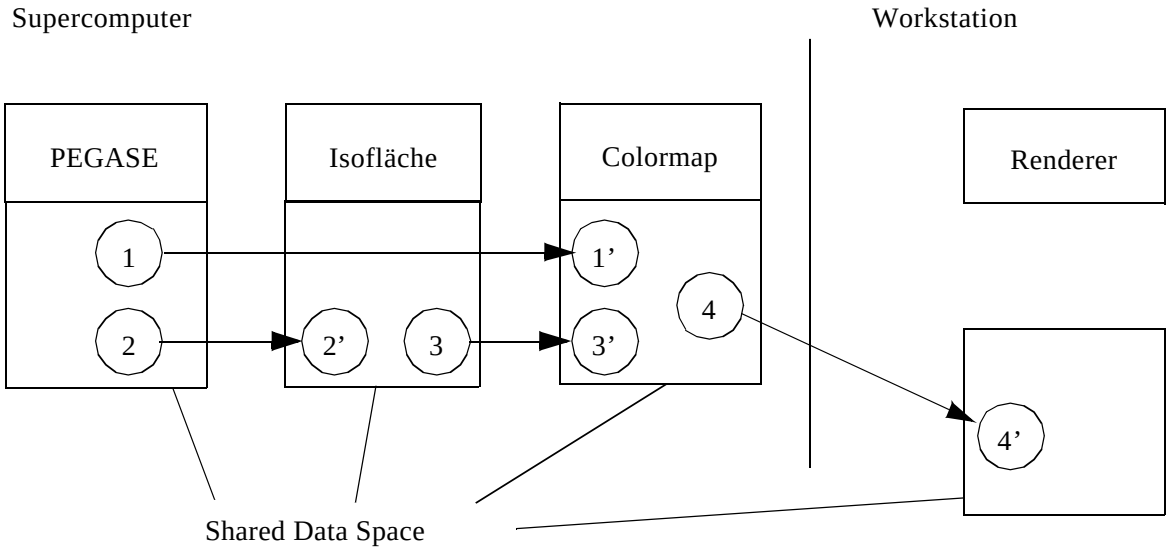

Abbildung 5-1: Der Datenfluß bei der Darstellung der PEGASE-Ergebnisse

In der linken Hälfte sind die Module zu sehen, die auf dem Supercomputer laufen. Da dort kein Shared Memory zur Verfügung steht, hat jeder Modul seinen eigenen Shared Data Space. Dies hat zwar (wie man auch an den Objekten erkennen kann) zur Folge, daß Daten kopiert werden müssen, obwohl beide Module auf demselben Rechner laufen; dies ist in Anbetracht der großen Datenmengen jedoch immer noch effizienter, als sie komplett über das Netzwerk zu schicken. Der PEGASE Modul schreibt nun das Geschwindigkeitsfeld (2) und die Wirbelstärke (1) in seinen Shared Data Space. Der Isoflächenmodul und der Colormapmodul greifen auf die Kopien dieser Datenobjekte zu (2’ und 1’). Diese Kopien werden jeweils von dem Requestbroker, der in den entsprechenden Modul integriert ist, angelegt, nachdem der Modul nach der Adresse dieses Datenobjektes gefragt hat. Das Ergebnisobjekt des Isoflächenmoduls (3) wird dann vom Colormapmodul benutzt, um die endgültig darzustellende Geometrie zu generieren (4). Damit der Rendermodul auf der Workstation anschließend das Bild erzeugen kann, muß nur dieses Geometrieobjekt in den Shared Data Space auf der Workstation kopiert werden (4’).

5.1.2 Strömung im Zylinder eines Motors

In diesem deutlich komplexeren Beispiel soll die kollaborative Visualisierung einer Off-Line gerechneten Simulation der Strömungsvorgänge in einem Verbrennungsmotor dargestellt werden (siehe Abbil-

dung 5-2). Das Szenario sei dabei bewußt komplex gewählt, um die Vorteile des Systems deutlich zu machen. (Das Beispiel der Strömung in einem Verbrennungsmotor ist dem ESPRIT-Projekt HPS-ICE entnommen.)

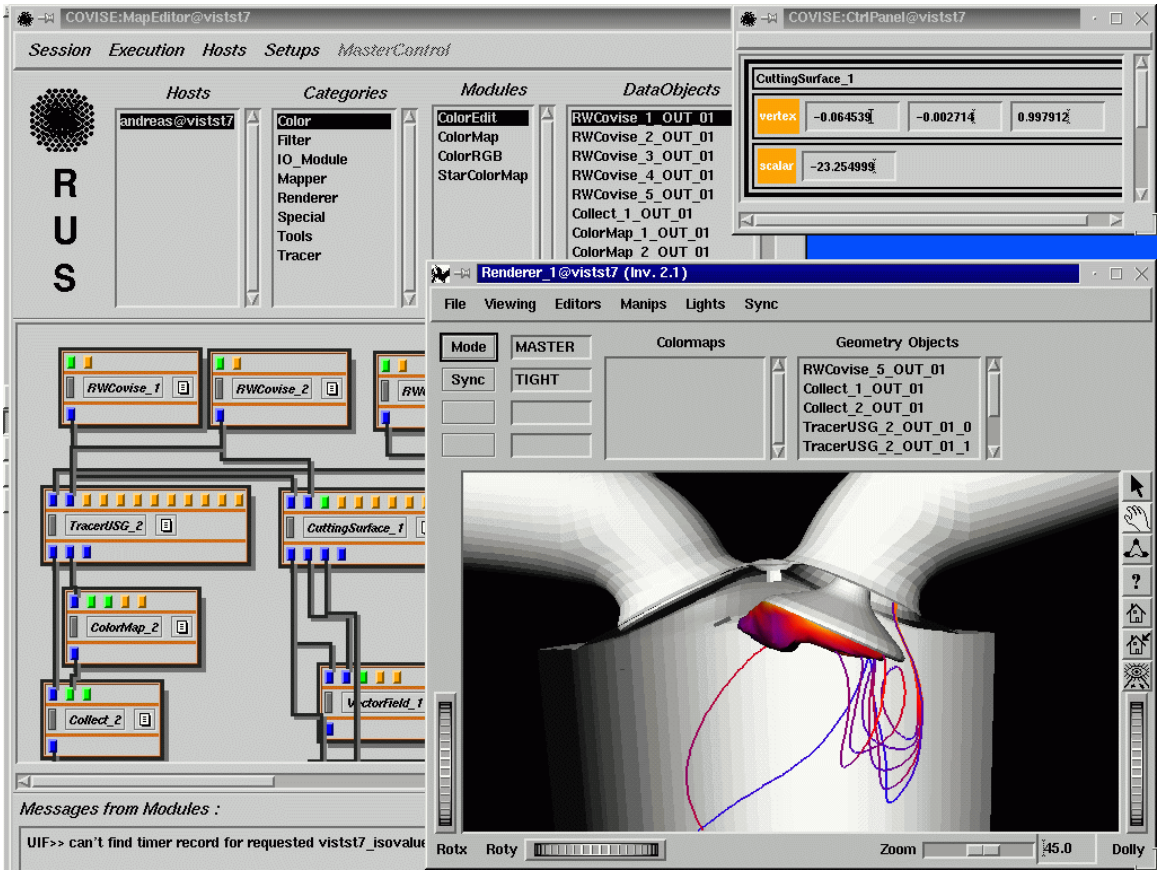

Abbildung 5-2: Bildschirmabzug des Motordatensatzes in der COVISE-Umgebung

Die verwendete Rechnerkonfiguration ist in Abbildung 5-3 zu sehen. Dort werden drei Rechner eingesetzt: ein Fileserver (F) und zwei Benutzerworkstations (A und B), wobei Benutzerworkstation B mittels einer relativ langsamen Netzwerkverbindung mit den beiden anderen Rechnern verbunden ist, deren

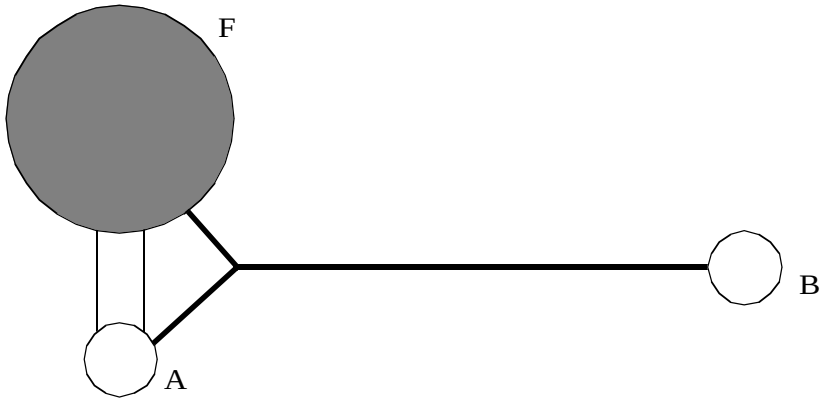

Abbildung 5-3: Leistungsfähigkeit von Rechnern und Netzwerk

Kosten sich auch nach der übertragenen Datenmenge richtet. Die Verbindung zwischen A und F soll mit gewöhnlicher LAN Geschwindigkeit arbeiten. Beide Workstations haben nur begrenzten Speicherplatz, der es nicht ermöglicht, den kompletten Datensatz vom Fileserver dorthin zu laden. Der Fileserver ist eigentlich nur für die Haltung von Daten, aber nicht für die Durchführung von Berechnungen gedacht. Dennoch steht eine, verglichen mit den Workstations, sehr hohe Rechenleistung zur Verfügung, die allerdings nur gegen eine entsprechend hohe Gebühr auch genutzt werden kann. In Abbildung 5-3 ist eine graphische Darstellung dieser Verhältnisse zu sehen, wobei Größe bzw. Dicke für die Leistungsfähigkeit des Rechners bzw. des Netzes stehen und die Farbe für die damit verbundenen Kosten (dunkel = teuer, weiß = preiswert).

Für die Visualisierung sollen nun folgende Algorithmen zum Einsatz kommen:

- die Berandung des Berechnungsgebietes muß bestimmt werden,
- um einen guten Eindruck von der Strömung zu bekommen, sollen anfangs einige Partikel verfolgt werden,
- zur genauen Analyse wird im Laufe der Sitzung mehrfach die Isofläche mit verschiedenen Werten für den Anteil des verbrannten Gases berechnet werden.

Dabei reicht die Speicherkapazität der beiden Workstation nur für die von der Isoflächenberechnung benötigten Daten aus. Für dieses Szenario soll jetzt eine optimale Verteilung der Visualisierung gefunden werden: die Visualisierung soll natürlich so schnell wie möglich, aber nur so teuer wie nötig sein.

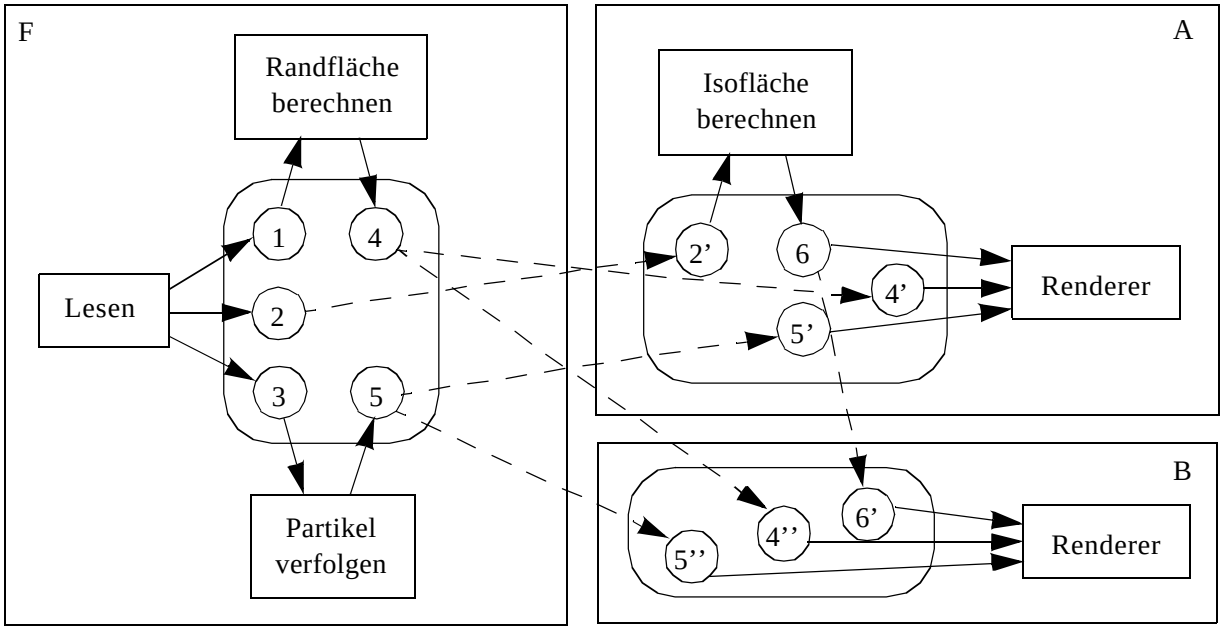

Abbildung 5-4: Verteilung der Module und Daten

Da es sich bei den Partikelverfolgungen um unter Umständen recht aufwendige Berechnungen handelt, die jedoch nur einmal vorkommen, können diese ohne allzugroße Kosten am besten direkt auf dem Fileserver erfolgen. Genauso wird mit der Berechnung der Randfläche verfahren. Die Berechnung der Isoflächen wird dagegen relativ häufig ausgeführt, so daß sie günstiger auf Rechner A ausgeführt werden kann, selbst wenn dies anfänglich einige Sekunden Zeit kostet, um die Daten vom Fileserver dorthin zu kopieren. Auf Workstation B muß nur der abschließende Renderschritt ausgeführt werden, kollaborativ

mit dem Renderer von Workstation A. Für die Verteilung ergibt sich die Situation wie sie in Abbildung 5-4 dargestellt ist.

Die Visualisierung läuft nun folgendermaßen ab: der Lesemodul liest die Daten aus einer Datei und schreibt sie in die entsprechenden Datenobjekte: Geometrie/Gitter (1), Restgaskonzentration (2) und Geschwindigkeitsfeld (3). Aus (1) wird nun direkt die Randfläche berechnet, aus (3) die Partikelbahnen. Der nächste Schritt findet nun auf Workstation A statt: die Isofläche fordert Objekt (2) an, dieses wird vom Request Broker zu Objekt (2') über die LAN-Verbindungen kopiert und der Isoflächenmodul erzeugt anschließend Datenobjekt (6). Der Renderer fordert abschließend die Objekte (4) und (5) an, die ebenfalls über die LAN-Verbindung vom Request Broker kopiert werden. Daraufhin sieht der Benutzer an Workstation A das vollständige Bild.

Parallel zur Isoflächenberechnung hat bereits der Renderer von Workstation B die Datenobjekte (4), (5) und (6) angefordert. Die Datenobjekte (4) und (5) werden bereits über die langsame WAN-Verbindung vom Request Broker in die Objekte (4'') und (5'') kopiert. Wenn diese Übertragung abgeschlossen ist und die Isoflächenberechnung auf Workstation B beendet ist, wird das Datenobjekt (6) von dort in das Objekt (6') auf Workstation B kopiert und der Renderer dort kann nun ebenfalls das komplette Bild zeigen.

Insgesamt werden auf dem teureren Fileserver die Berechnungen der Randfläche und der Partikelverfolgung durchgeführt. Über die langsame Netzwerkverbindung werden die Randfläche, die Ergebnisse der Partikelverfolgung und die Isofläche übertragen.

Soll nun die Isofläche neu berechnet werden, so ergibt sich folgender Ablauf: der Isoflächenmodul auf Workstation A greift auf das Datenobjekt (2) zu. Da bereits eine Kopie (2') dieses Datenobjektes im lokalen Shared Data Space existiert, wird dessen Adresse zurückgegeben und der Isoflächenmodul kann seine Arbeit beginnen, ohne daß Daten über das Netz geschickt werden müssen. Wenn der Isoflächenmodul fertig ist, schreibt er ein neues Datenobjekt (6) und die beiden Renderer auf den Workstations A und B werden über diese Änderung informiert. Der Renderer auf Workstation A greift auf die Objekte (4'), (5') und (6) direkt zu und stellt das neue Ergebnis dar. Auf Workstation B greift der Renderer auf die Objekte (4'') und (5'') direkt zu, während vor dem Zugriff auf das neue Objekt (6) eine neue Kopie (6') angelegt werden muß. Diese Kopie wird durch Übertragung aus dem Shared Data Space von Workstation A erzeugt.

Für die Darstellung der geänderten Isofläche ist nur eine kostenpflichtige Übertragung der Geometrie dieser Isofläche über die WAN-Verbindung erforderlich. Ansonsten findet keinerlei Datentransfer statt. Dies macht deutlich, daß der in dieser Arbeit dargestellte Ansatz auch in der Lage ist, für komplexeste, im Industriealltag tatsächlich vorkommende Problemstellungen, eine optimale Lösung anzubieten.

In Abbildung 5-5 ist die hierarchische Struktur des Aufbaus der Datenobjekte anhand der Partikelbahn dargestellt. Das Hauptobjekt ganz links ist vom Typ Geometrie: dieser beinhaltet die reine Geometrieinformation, eventuelle Normalen, Farben und Texturen. In diesem Beispiel sind nur die Geometrie und die Farben präsent. Das erste Unterobjekt der Geometrie ist eine Menge von Linien: eine Partikelbahnberechnung hat hier fünf Partikelbahnen generiert, die in einem Set-Objekt zusammengefaßt sind (analog dazu ist auch das Unterobjekt für die Farben (hier nicht explizit dargestellt) ein Set-Objekt das fünf Elemente mit den entsprechenden Felder enthält). Jede Partikelbahn besteht aus einer Linie (zweites Objekt von rechts), die wiederum aus Punkten zusammengesetzt ist (Objekt ganz rechts). In der Darstellung ist auch zu erkennen, daß zu jedem Objekt Attribute gesetzt werden können. In diesem Fall sind

dies die Farbe der Partikelbahn sowie eine Feedback-Information, die es einem Renderer erlaubt, die Position des Startpunktes direkt zu beeinflussen.

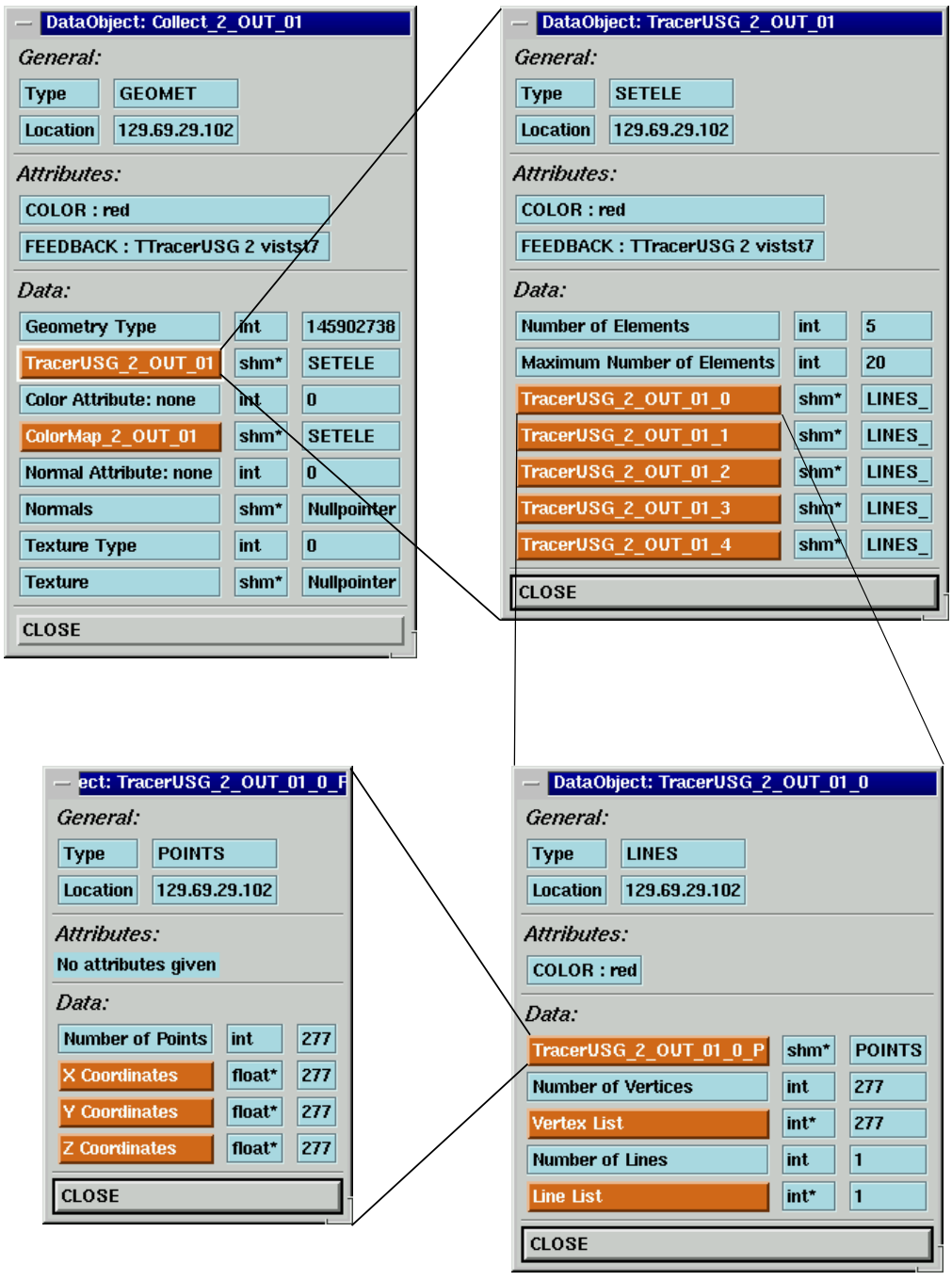

Abbildung 5-5: Datenobjekt-Anzeige der Partikelbahn

5.1.3 Strömung in einem Saugrohr

In diesem Fall wird die numerische Berechnung für die Simulation der Wasserströmung im Saugrohr einer Turbine auf einem Supercomputer durchgeführt (siehe Abbildung 5-6). Die Module zur Visualisierung dieser Strömung (Schnittfläche mit Vektorpfeilen, Partikelbahnen, Isofläche einer Geschwindig-

keitskomponente) laufen entweder ebenfalls auf dem Supercomputer oder aber auf der Visualisierungsworkstation. Der Benutzer hat über die Benutzerschnittstelle der Graphikworkstation die Möglichkeit, den Eintrittswinkel der Strömung in das Saugrohr zu bestimmen. Je nach Winkel bildet sich dann in der Strömung eine Rückströmzone. Da die stationäre Berechnung der Strömung auf dem Supercomputer eine relativ kurze Rechenzeit benötigt, können die Ergebnisse der ersten Iterationen bereits nach wenigen Sekunden visualisiert werden.

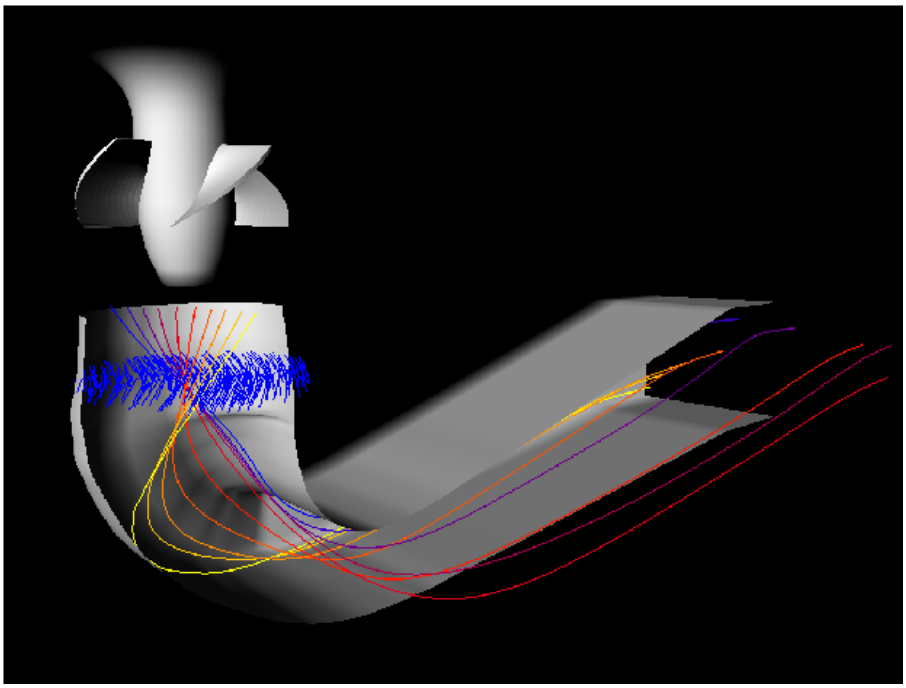

Abbildung 5-6: Strömung im Saugrohr einer Wasserturbine

Weitere Anwendungen

Die COVISE-Software wird in zahlreichen Projekten des Rechenzentrums weiterentwickelt oder dient als Basis für andere Entwicklungen. Zu diesen Projekten gehören unter anderen

- das vom DFN-Verein finanzierte Projekt NumLab:
  hier wurde in Zusammenarbeit mit dem Institut für Angewandte Mathematik der Universität Freiburg eine kollaborative Umgebung entwickelt, die es verteilt sitzenden Nutzern von Parallelrechnern ermöglicht, ihren Numerikcode mit Hilfe des vor Ort sitzenden Spezialisten zu parallelisieren, ihn auf dem Parallelrechner laufen zu lassen und die Ergebnisse anschließend zu visualisieren.
- das vom DFN-Verein finanzierte Projekt VRS-Lab:
  mit dem Max-Planck-Institut für Biokybernetik und dem Institut für Theoretische Astrophysik in Tübingen (TAT) wird eine verteilte Umgebung für eine kollaborative Visualisierung mittels Techniken der virtuellen Realität entwickelt.
- das BMBF geförderte Projekt EFENDA:
  in der deutschen Luft- und Raumfahrtindustrie wird COVISE als Werkzeug für die Integration von Numerik und Visualisierung in einer verteilten und kollaborativen Umgebung etabliert.
- das G7 Referenzprojekt SPOCK:

das Szenario einer verteilten Umgebung wird hierbei auf den experimentellen Bereich ausgeweitet: die Ergebnisse von Windkanalversuchen und Freiflugexperimenten werden über Module in die COVISE-Umgebung eingebracht und gemeinsam visualisiert

- das ESPRIT-Projekt COVAS:
  die Funktionalität von COVISE wird den Bedürfnissen der Automobilindustrie angepaßt und mit einem regionalen und einem internationalen Partner in Hinblick auf seine Effizienz im Industriealltag hin untersucht.
- das ESPRIT-Projekt VISiT:
  COVISE wird als Integrationsplattform eingesetzt, um den gesamten Zyklus von der Gittergenerierung über die Berechnung bis hin zur Analyse der Ergebnisse aus einer einzigen VR-Umgebung heraus handhabbar zu machen.

In allen diesen Projekten kommt die in dieser Arbeit dargestellte Datenverwaltung zur Anwendung. Dabei werden in den verschiedenen Projekten die Schwerpunkte durchaus auch auf verschiedene Stärken dieses Ansatzes gelegt: bei SPOCK auf die Erweiterbarkeit, bei COVAS auf die Flexibilität auch bei der Visualisierung großer Datenmengen über relativ langsame Datenverbindung, bei VRS-Lab auf die effiziente Datenübertragung bei Echtzeit-relevanten Anwendungen. Die Datenverwaltung, neben der Gesamtsteuerung der zentrale Teil in COVISE, erfüllt dabei höchste Anforderungen unter verschiedensten Bedingungen. Siehe auch [3] [37] [70] und [79].

5.2 Messungen

In Kapitel Kapitel 4, Konzept eines verteilten Visualisierungssystems, wurde bereits mehrfach erwähnt, daß bei der Konzeption und Implementierung großer Wert auf eine effiziente Ausnutzung von High-Performance-Komponenten wie Supercomputern oder schnellen Netzwerken gelegt wurde (vergleiche hierzu auch [22] [47] und [77]). In diesem Kapitel soll anhand einiger ausgewählter Messungen belegt werden, daß dieses Ziel erreicht wurde. Der Vergleich mit anderen verteilten Visualisierungswerkzeugen gestaltet sich jedoch etwas schwieriger: die Meßpunkte, die für die Messungen benutzt wurden liegen zu einem großen Teil an Stellen des Systems die bei kommerziellen Produkten für den Modulprogrammierer nicht zugängig sind. Daher können für diese Produkte nur wenige Messungen durchgeführt werden.

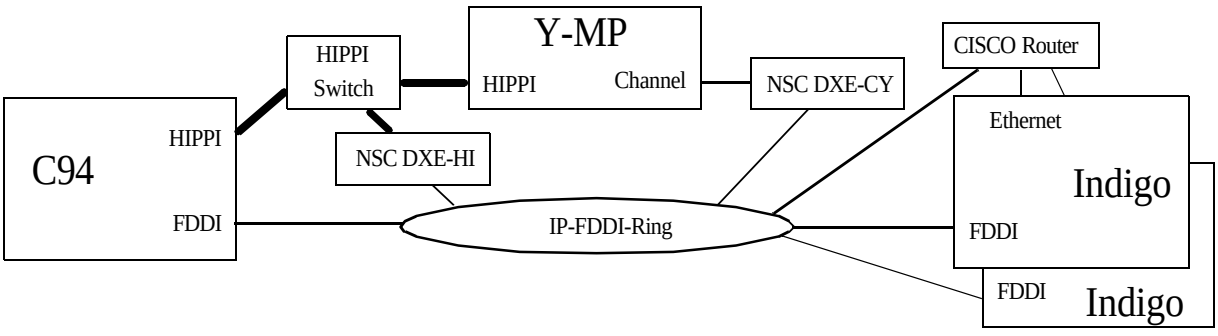

Abbildung 5-7: Umgebung für die Messungen

In Abbildung 5-7 ist ein Ausschnitt aus der Konfiguration des Rechenzentrums zu sehen, auf der die Messungen dieses Kapitels zum größten Teil durchgeführt wurden. Gemessen wurden die Datenraten zwischen folgenden Rechner/Netzwerkinterface-Paaren:

- C94 / HIPPI und Y-MP / HIPPI (direkt über den HIPPI Switch)
- C94 / FDDI und Y-MP / FDDI (über den NCS DXE-CY Channel-Router)
- Y-MP / HIPPI und Indigo / FDDI (über den HIPPI-Switch und den NXC DXE-HI Router)
- C94 / FDDI und Indigo / FDDI (direkt über den FDDI-Ring)
- C94 / FDDI und Indigo / Ethernet (über den CISCO-Router)
- Indigo / FDDI und Indigo / FDDI (direkt über den FDDI-Ring)

Zu bemerken ist noch, daß der FDDI-Ring physikalisch aus zwei FDDI-Ringen besteht, die über einen Giga-Switch gekoppelt sind. Logisch handelt es sich jedoch um einen einzigen Ring. Der Durchsatz wird durch den zusätzlichen Router jedoch nur minimal (d.h. kaum meßbar) beeinträchtigt, so daß es durchaus legitim ist, von einem FDDI-Ring zu sprechen.

Die Rechner in dieser Abbildung sind folgendermaßen konfiguriert:

- Cray C94
  4 Prozessoren, 8 GB Hauptspeicher, UNICOS 8.0.2.3, HIPPI und FDDI Netzwerkinterface, normalerweise voll ausgelastet während der Messungen (mehr als 25 numerische Simulationen gleichzeitig).
- Cray Y-MP 2E
  2 Prozessoren, 256 MB Hauptspeicher, UNICOS 8.0.2.3, HIPPI Netzwerkinterface und FDDI-Channel Router, dedizierter Fileserver (keine numerischen Simulationen, ~ 60 nfsd, zahlreiche Migrationsprozesse), mit 10 GB schnellen Platten (~50 MB/s) und mehr als 2 TB On-line Bandspeicher. Der Hauptspeicher ist sehr knapp, was zu einer heftigen Swap-Aktivität führt.
- SGI Indigo
  100 MHz R4000 Prozessor, 128 MB Hauptspeicher, XS24Z Graphik, IRIX 5.3, Ethernet und FDDI Netzwerkinterface, Single User Workstation.

Da das Rechenzentrum eine höchstmögliche Verfügbarkeit seiner Services den Benutzern gegenüber anstrebt, mußten die Messungen unter ganz normalen Bedingungen stattfinden. D.h. die beiden Cray-Rechner waren unter voller Last: die C94 in der Regel mit 25 und mehr Numerikprozessen (wobei durch den mit 8 Gigabyte sehr großen Hauptspeicher in den meisten Fällen kein Mangel an Speicher besteht und daher auch kein Prozeß auf Festplatten ausgelagert werden mußte), die Y-MP hatte 60 und mehr aktive NFS-Prozesse und war mit 256 Megabyte Hauptspeicher sehr knapp ausgestattet. Da die Messungen unter Betriebsbedingungen zum Teil sehr schlechte Ergebnisse zeigten, durfte für eine Folge von Messungen die Wartungszeit der beiden Cray-Rechner genutzt werden. Zu dieser Zeit (die leider nur recht selten und relativ kurz ist) befanden sich nur noch zwei aktive Systemadminstratoren auf den Rechnern, die keinerlei CPU-intensiven Programme laufen ließen. Bei diesen Messungen, die zwischen jeweils einer Cray und einer Indigo stattfanden, wurde der weiter im Normalbetrieb befindliche FDDI-Ring genutzt; dieser war jedoch aufgrund der Wartung der beiden Cray-Rechner nahezu ungenutzt.

Ziel der Messungen war, genauen Aufschluß darüber zu bekommen, welche Stellen in diesem verteilten Datenmanagement kritisch für die Gesamtperformance des Systems sind. Dazu genügt es nicht, einfach die Zeit zwischen der Anforderung eines Datenobjektes, das auf einem anderen Rechner liegt, und seiner Verfügbarkeit zu messen. Vielmehr sollten so viele der sich hinter diesem Vorgang verbergenden Einzelschritte so genau wie möglich erfaßt werden, um auch die Stellen herauszuarbeiten, an denen weitere Optimierungen den größten Erfolg versprechen.

5.2.1 Messungen zwischen zwei Indigo Workstations

Es handelt sich hierbei um zwei identisch konfigurierte Silicon Graphics Indigo XS24. Für diese Messungen wurde ein Modul benutzt, der eine Partikelverfolgung durchführt (Tracer). Dieser Tracer greift auf ein Datenobjekt mit einer Größe von 3,182,608 Bytes zu, das sich auf dem anderen Rechner befindet. Daraus ergibt sich die folgende Reihenfolge der Aktionen (in Klammern stehen die Nummern der Meßpunkte in Abbildung 5-8): zunächst fragt der Tracer-Modul seinen lokalen Request Broker nach dem gewünschten Objekt (1 und 2). Dieser sucht in seinen lokalen Datenstrukturen, stellt fest, daß sich dieses Datenobjekt nicht in seinem Shared Data Space befindet (2 und 3) und bittet daraufhin den Request Broker der Indigo B um Zusendung dieses Objektes (3 und 4). Request Broker B sucht das Datenobjekt in seinen Strukturen und bereitet es für den Transfer vor (4 und 5).

Diese Vorbereitung besteht im wesentlichen aus dem Einpacken des Datenobjektes: aus der Repräsentation des Datenobjektes im Shared Data Space wird es Element für Element in einen Puffer kopiert, der dann den Systemfunktionen für das Senden von Daten über einen Socket übergeben wird. Die nächsten beiden Meßpunkte beziehen sich auf das Senden selbst. Nach Meßpunkt 5 fängt der Request Broker B an zu senden und Meßpunkt 6 wird erreicht, nachdem die zu übertragenden Daten komplett bei Request Broker A angekommen sind. Dieser muß dann wiederum aus den Daten im Puffer ein Datenobjekt im Shared Data Space erzeugen (6 und 7). Als letztes wird dann die Adresse des Datenobjektes wieder an den Tracer-Modul, der das Datenobjekt angefordert hat, zurückgeschickt; dieser baut dann sein lokales Zugriffsobjekt auf (siehe Abschnitt 4.2.3, Zugriff eines Moduls auf die Datenobjekte, 7 und 8).

Die einzelnen Meßpunkte sind dabei an den folgenden Stellen im Quellcode plaziert:

1. vor dem Aufruf des Konstruktors für das benötigte Datenobjekt
2. direkt bei der Ankunft der Botschaft
3. unmittelbar vor dem Absenden der Botschaft an den Request Broker B
4. direkt bei der Ankunft der Botschaft
5. unmittelbar vor dem Absenden der Botschaft an Request Broker A
6. direkt bei der Ankunft der Botschaft
7. unmittelbar vor dem Absenden der Botschaft an Request Broker A
8. nach dem Aufruf des Konstruktors

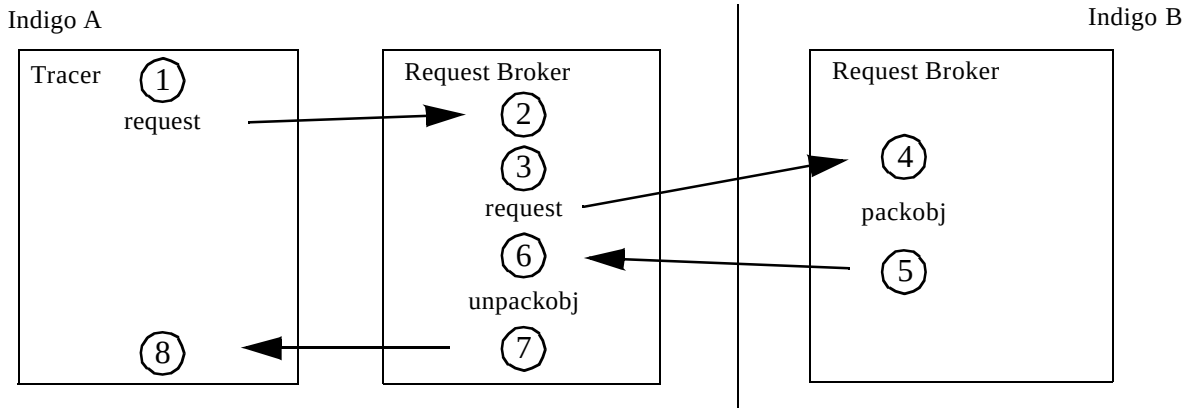

Abbildung 5-8: Meßpunkte der Konfiguration Indigo/Indigo

Tracer	lokaler RB	Warten in lokalem RB	remote RB	Netzwerk	Netto-Durchsatz	Brutto-Durchsatz
8 - 1	7 - 2	6 - 3	5 - 4	(6 - 3) - (5 - 4)	Größe / Netzwerk	Größe / Tracer
1.006	0.981	0.737	0.292	0.445	7151 KB/s	3163 KB/s
0.733	0.726	0.592	0.187	0.404	7871 KB/s	4338 KB/s
0.601	0.594	0.458	0.159	0.299	10627 KB/s	5293 KB/s
0.616	0.609	0.466	0.158	0.308	10332 KB/s	5164 KB/s
0.597	0.591	0.456	0.158	0.297	10698 KB/s	5323 KB/s
0.606	0.599	0.460	0.160	0.300	10599 KB/s	5248 KB/s
0.690	0.682	0.543	0.159	0.384	8277 KB/s	4599 KB/s
0.596	0.590	0.454	0.158	0.295	10752 KB/s	5333 KB/s

Tabelle 5-1: Indigo zu Indigo, FDDI

Zwar ließe sich nach Meßpunkt 1 und vor Meßpunkt 8 noch jeweils ein Meßpunkt analog zu den Meßpunkten 3 und 6 einschieben, aus Gründen des Aufwandes und der Übersichtlichkeit und da das eigentliche Interesse ja der Kommunikation der Request Broker untereinander gilt, wurde darauf verzichtet.

In Tabelle 5-1 sind die Ergebnisse von acht Messungen zu sehen. In der zweiten Zeile sind jeweils die Meßpunkte zu sehen, zwischen denen die entsprechenden Werte ermittelt wurden. In den beiden letzten Spalten befinden sich die Ergebnisse einer Netto- und einer Brutto-Durchsatzberechnung. Für den Netto-Durchsatz wurde die Datenobjektgröße (3,182,608) geteilt durch die Zeit, die im Netzwerk verbracht wurde (Spalte fünf). Dies ergibt eine Zahl, die hinter dem eigentlich erreichten Durchsatz bei der Übertragung des Datenobjektes (6 - 5) zurückbleibt (der Netto-Durchsatz beschreibt das Mittel der beiden Übertragungen 4 - 3 und 6 - 5). Die Übertragungszeit zwischen 3 und 4 dürfte in der Größenordnung der Hälfte der Differenz zwischen Spalte eins und Spalte zwei liegen. Der Brutto-Durchsatz wurde mit der Gesamtzeit, die der Tracer Modul auf das Datenobjekt warten mußte, (Spalte eins) berechnet.

Statt der relativ konservativ berechneten Netzwerkzeit wäre natürlich die genaue Übertragungsrate für eine einzelne Botschaft interessanter. Allerdings wäre für diese Art von Messungen eine absolute Übereinstimmung der Uhren der beteiligten Rechner nötig. Diese Übereinstimmung war jedoch selbst bei den beiden sehr eng gekoppelten Cray-Supercomputern nicht gegeben. Auch durch vergleichende Messungen der Hin- und Rückübertragung ließen sich nie die exakten Werte ermitteln, so daß bei diesen Messungen nur Meßpunkte zueinander in Beziehung gesetzt werden, die auf demselben Rechner mit derselben Uhr gemessen wurden. Etwaige Korrekturen dieser Uhren während einer laufenden Messung können nicht ausgeschlossen werden, dürften sich aber im Millisekundenbereich oder darunter bewegen; bei mehreren Messungen ist es auch hinreichend unwahrscheinlich, daß diese Sprünge immer an der gleichen Stelle auftreten. Da bei diesen Messungen auch nicht die maximal erreichbaren Leistungen der Netzwerkinterfaces des Workstationherstellers von Interesse sind, sondern vielmehr die Zeit, die insgesamt im Netzwerk verloren geht, reichen die Meßpunkte absolut aus, um einen Eindruck von der Leistungsfähigkeit dieses Ansatzes zu geben. Bei der Berechnung der Netzwerkzeit treten eventuelle Unterschiede zwischen den Rechnerzeiten und der genauen Zeit nicht auf (sie heben sich auf). Die einzig möglichen Differenzen könnten aus einem verschieden schnellen Laufen der Uhren der beteiligten Rechner kommen, was aber beim Stande der heutigen Technik in den Größenordnungen dieser Messungen getrost vernachlässigt werden kann.

Betrachtet man die Zeit, die der Tracer insgesamt warten mußte (Spalte eins) so sieht man, daß sich die optimale Performance erst nach ein oder zwei Durchläufen einstellt (bedingt durch Initialisierungen, die nur beim ersten Mal auftreten oder eine vom Betriebssystem verursachte Optimierung von Cache-Inhalten).

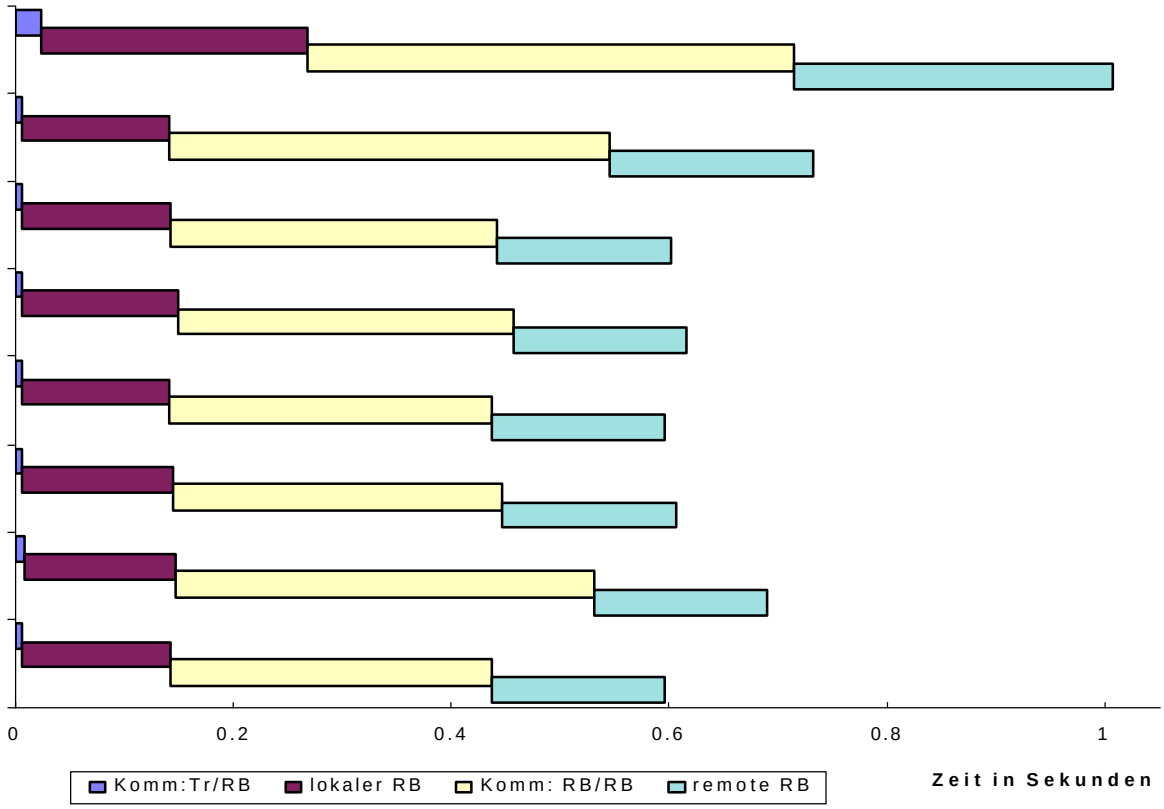

Abbildung 5-9: Graphik der Meßergebnisse aus Tabelle 5-1

Die Grafik in Abbildung 5-9 zeigt, daß sich der Zeitunterschied fast gleichmäßig auf die verschiedenen Teile der Messungen verteilt. Interessant ist, daß eine solche „Initialisierungsbremse“ bei keiner der anderen Messungen in dieser Form auftritt. Nach dieser Anfangsphase zeigen die Messungen jedoch absolut gleichbleibende Werte; nur die vorletzte Messung ist etwa eine Zehntelsekunde langsamer als die anderen, was sich aber eindeutig auf das Netzwerk zurückführen läßt.

Insgesamt ergibt sich eine ausgesprochen gleichmäßige Verteilung der Zeiten auf das Netzwerk selbst und die Vor- und Nachbereitungszeit. Selbst unter idealen Netzwerkbedingungen (d.h. unendliche Netzgeschwindigkeit) ließe sich maximal ein Bruttodurchsatz von etwa 11 Megabyte pro Sekunde erreichen. Ebenfalls interessant ist die Zeit, die für die Kommunikation zwischen dem Tracer-Modul und dem Request Broker gebraucht wird. Sie liegt mit etwa 7 ms bei einem Prozent der Gesamtlaufzeit. Bei einer Vorwegübertragung von Datenobjekten (d.h. Datenobjekte, von denen der Controller weiß, daß sie auf einem anderen Rechner gebraucht werden, werden sofort, nachdem der erzeugende Modul seine Schreibrechte zurückgegeben hat, an den entsprechenden Request Broker übertragen) sind diese 7 ms der Gesamtoverhead, der durch die Verwendung dieses Konzeptes bedingt ist. Dieser Overhead ist natürlich konstant, so daß bei kleineren Datenobjekten die Auswirkungen größer sein werden, bei größeren dagegen geringer.

5.2.2 Messungen zwischen den beiden Cray Vektor-Supercomputem

Da die beiden Cray-Rechner nicht über Shared Memory verfügen, liegen die Meßpunkte bei diesen Messungen etwas anders als bei den reinen Workstation-Messungen. Bei dem das Datenobjekt (das in diesem Fall eine Größe von 15,667,376 Bytes hat) anfordernden Modul handelt es sich um einen Cuttingplane Modul, der mit seinem Request Broker in ein einziges Executable integriert ist. Dadurch steht anstelle der Kommunikation zwischen den Meßpunkten 1 und 2 sowie 7 und 8 aus Abbildung 5-8 nur ein Funktionsaufruf. Daher sind die Meßpunkte 2 und 7 bei diesen Messungen weggelassen worden. Der Request Broker auf der C94 ist ebenfalls Teil eines Moduls, nämlich des PEGASE-Moduls der die Daten

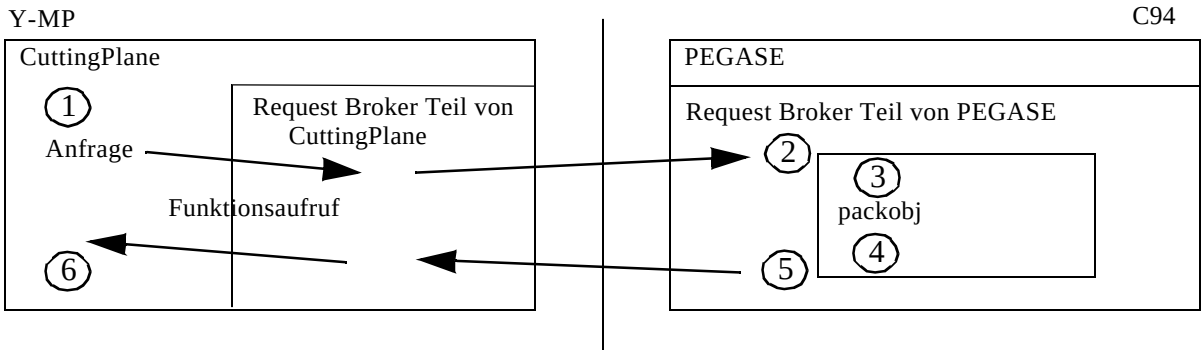

Abbildung 5-10: Meßpunkte der Konfiguration Y-MP/C94

berechnet. An Stelle der Meßpunkte 3 und 6 des lokalen Request Brokers sind hier die Meßpunkte 3 und 4 auf dem PEGASE-Request Broker zusätzlich direkt um das Einpacken des Datenobjektes gruppiert. Dadurch läßt sich der Aufwand für das Suchen des Datenobjekts in den Datenstrukturen des Request Brokers abschätzen. Dies ist jedoch für die Messungen nicht von Belang. In Abbildung 5-10 sind die hier verwendeten Meßpunkte zu sehen, deren Plazierung im Quellcode den Meßpunkten aus Abbildung 5-8 entspricht.

FDDI-Verbindung

In Tabelle 5-2 sind die Ergebnisse dieser Messungen zu sehen. Auffallend an diesen Ergebnissen ist, daß sie deutlich unter den Erwartungen für Messungen mit Supercomputern liegen.

Eine maximale Transferrate von etwas über 2 MB/s über FDDI ist insbesondere im Vergleich zu den Messungen zwischen zwei mit FDDI vernetzten Workstations (Abschnitt 5.2.1) äußerst unbefriedigend. Hinzu kommt die Schwankung der Meßwerte bis hinunter zu 200 KB/s. Verursacht werden diese schlechten Durchsatzraten allein durch die Spalte Netzwerk.

CuttingPlane	DM in PEGASE	packobj	Netzwerk	Netto Durchsatz	Brutto Durchsatz
6 - 1	5 - 2	4 - 3	(6 - 1) - (5 - 2)	Netzwerk / Größe	CuttingPlane / Größe
7.448	0.015	0.015	7.432	2107 KB/s	2103 KB/s
74.945	0.015	0.014	74.930	209 KB/s	209 KB/s
14.738	0.016	0.016	14.721	1064 KB/s	1063 KB/s
19.228	0.016	0.015	19.211	815 KB/s	815 KB/s

Tabelle 5-2: C94 zu Y-MP, FDDI

Eine genauere Untersuchung förderte folgende Ursachen für diese Meßergebnisse zu Tage: der bei der Verbindung zwischen den beiden Crays verwendete Gigaswitch zeigt aufgrund hoher Belastung durch viele Zugriffe auf den Fileserver (die Y-MP) einen hohen Packetverlust durch Blockierung seiner Puffer (Head-of-Line-Blocking). Selbst eine direkte Low-Level-Messung ergab nicht mehr als 4MB/s für diese Verbindung. Hinzukommt, daß die Y-MP nur sehr knapp mit Hauptspeicher ausgestattet ist. Dadurch werden Prozesse, die warten (was der das Datenobjekt anfordernde CuttingPlane-Modul tut) sehr schnell aus dem Hauptspeicher in den Swapbereich ausgelagert. Das Wiedereinlagern dauert dann unter Umständen sehr lange (daher die über 70 Sekunden im zweiten Durchgang). Diese Probleme sind inzwischen durch Umkonfiguration gelöst worden. Die sehr geringen Unterschiede zwischen der zweiten und dritten Spalte belegen auch den minimalen Overhead, den das Suchen des Datenobjektes in den Datenstrukturen des Request Brokers mit sich bringt.

HIPPI-Verbindung

Für diese Messungen gelten die gleichen Meßpunkte wie im Fall der FDDI-Verbindung. Die im Vergleich höheren Zeiten für die im Request Broker verbrachte Zeit ist auf die zum Zeitpunkt der Messungen höhere Auslastung der C94 zurückzuführen (es handelte sich bei diesen Messungen um exakt den gleichen Datensatz). Der größte Teil der Zeit wird immer noch im Netzwerk verbracht. Selbst der Netto-Durchsatz ist weit von den theoretisch möglichen 100 MB/s entfernt. Die zu diesem Zeitpunkt geringere Auslastung der Y-MP führt dazu, daß nun direkt der durch das Scheduling der C94 bedingte Zeitverlust sichtbar ist. Zwischen der Beendigung der Arbeit des PEGASE-Moduls und der Anforderung des Datenobjekts durch den Request Broker des CuttingPlane Moduls auf der Y-MP wird der Prozeß einfach in einen Wartezustand versetzt. Aus diesem Wartezustand kommt er erst nach einigen Sekunden wieder zurück, so daß die Netzwerkzeit um diese Sekunden größer erscheint, als sie eigentlich ist.

CuttingPlane	RB in PEGASE	packobj	Netzwerk	Netto-Durchsatz	Brutto-Durchsatz
4.513	0.054	0.053	4.458	3513 KB/s	3471 KB/s
3.389	0.023	0.022	3.366	4653 KB/s	4621 KB/s
3.932	0.063	0.062	3.869	4048 KB/s	3983 KB/s
1.278	0.055	0.054	1.223	12804 KB/s	12251 KB/s

Tabelle 5-3: C94 zu Y-MP, HIPPI

Der erste Versuch, diese Wartezeiten zu reduzieren, ist die Nutzung des asynchronen write-Befehls der C94. Dieser Befehl unterscheidet sich von einem normalen write-Befehl im wesentlichen dadurch, daß er sofort zurückkommt und nicht erst, nachdem die Daten tatsächlich verschickt wurden. Normalerweise nutzt man diesen Befehl, um parallel zu dem dann vom Betriebssystem abgewickelten write das Programm weiterarbeiten zu lassen. In diesem Fall ist allerdings die Arbeit des Request Brokers mit dem Abschicken vorerst beendet. Der eigentliche Vorteil des asynchronen write liegt hier jedoch darin, daß, nachdem das asynchrone write zurückkommt, der eigentliche Sendevorgang vom Betriebssystem mit der Priorität des Kernels durchgeführt wird, während bei einem normalen write die Priorität des Benutzerprozesses die Ausführung bestimmt.

Die Ergebnisse in Tabelle 5-4 zeigen deutlich, daß die durchschnittliche Durchsatzrate deutlich über der des synchronen write liegt. Das heißt für den Benutzer zeigt sich eine sichtbar verbesserte Leistung, obwohl sich an der Handhabung für ihn nichts geändert hat. Durch die Nutzung des asynchronen `write` muß der Prozeß nun nicht mehr darauf warten, daß die Daten vollständig über das Netzwerk weggeschickt wurden, bevor die `write`-Routine wieder zurückkommt. Der maximal erreichbare Durchsatz liegt jedoch in der selben Größenordnung. Die Schwankungen in den Zeiten, die zum Einpacken des

Objektes benötigt werden, sind wiederum eine Folge der Auslastung der C94. An den optimalen Werten (3 ms) wird auch das tatsächliche Potential dieser Supercomputer deutlich: 3 ms zum Einpacken von mehr als 15 MB ergeben einen Durchsatz von etwa 490 MB/s. Im Vergleich dazu brauchen die Indigo Workstations aus Abschnitt 5.2.1 für etwa 3 MB mehr als 150 ms, was einem Durchsatz von etwa 20 MB/s entspricht. Hierbei kommen die Vorteile des auf optimalen Memory-Durchsatz optimierten Systems zur Geltung. Der Netzwerkdurchsatz bleibt jedoch immer noch hinter den Erwartungen zurück.

CuttingPlane	RB in PEGASE	packobj	Netzwerk	Netto Durchsatz	Brutto Durchsatz
1.282	0.059	0.030	1.222	12817 KB/s	12219 KB/s
1.754	0.003	0.002	1.751	8946 KB/s	8931 KB/s
1.610	0.003	0.002	1.607	9748 KB/s	9729 KB/s
3.164	1.657	1.656	1.507	10391 KB/s	4950 KB/s
1.852	0.003	0.002	1.849	8472 KB/s	8458 KB/s
2.047	0.016	0.016	2.030	7714 KB/s	7652 KB/s
1.557	0.016	0.015	1.541	10161 KB/s	10056 KB/s

Tabelle 5-4: C94 zu Y-MP, HIPPI, asynchrones write

Der Schlüssel zur besseren Ausnutzung der HIPPI-Infrastruktur liegt in den Parametern der verwendeten Sockets, hier im wesentlichen bei der Puffergröße. Die Standardpuffergröße, die vom Betriebssystem vorgegeben wird, ist für den Einsatz von Ethernet oder FDDI gedacht und liegt in der Größenordnung von mehreren KB. Da die Leistung von HIPPI mit seiner theoretischen Maximaltransferrate von 100 MB/s jedoch deutlich über der dieser Netzwerke liegt, wird der Overhead, der mit den Standardpuffergrößen verbunden ist, zu einer deutlichen Bremse: bei 100 MB/s und einer Puffergröße von 8 KB müssen ca. 12500 Pakete verschickt werden. Durch Erhöhen der Puffergröße auf 384 KB reduziert sich diese Anzahl auf etwa 300 Pakete. Der Parameter TCP_WINSHIFT sorgt unter UNICOS für eine entsprechende Vergrößerung des Puffers. In Tabelle 5-5 ist denn auch der Erfolg dieser Maßnahmen zu sehen.

CuttingPlane	RB in PEGASE	packobj	Netzwerk	Nettodurchsatz	Bruttodurchsatz
2.419	0.011	0.008	2.408	2672 KB/s	2660 KB/s
6.080	0.019	0.009	6.069	1060 KB/s	1058 KB/s
0.194	0.078	0.073	0.116	55475 KB/s	33171 KB/s
19.217	0.012	0.009	19.205	335 KB/s	334 KB/s
0.130	0.013	0.009	0.117	55001 KB/s	49501 KB/s
24.933	0.015	0.009	24.918	258 KB/s	258 KB/s
20.873	0.013	0.008	20.860	308 KB/s	308 KB/s

Tabelle 5-5: C94 zu Y-MP, asynchrones write, optimale Socket-Parameter

In den meisten der Durchläufe ist die Auslastung der beiden beteiligten Cray-Rechner ersichtlich. Insbesondere bei der Y-MP wird der anfragende Prozeß meist aus dem Hauptspeicher in den Swap-Bereich verlagert, bevor die Antwort wieder da ist. Aber im dritten und besonders im fünften Fall ist zu erkennen, daß ein deutlicher Anteil des theoretisch erreichbaren Durchsatzes realisiert wurde. Genaue Berechnungen des Nettodurchsatzes ergaben maximale Transferraten von etwa 66 MB/s (durch genaue Inspektion der Meßzeiten und Vergleich von Hin- und Rückweg). Speziell im fünften Durchlauf wird auch deutlich, daß der Benutzer einen sehr großen Anteil dieser Performance tatsächlich in der Anwendung sieht. Es ist also durchaus möglich, aus einer „Standard“-Anwendung heraus durch optimale Einstellung der Parameter einem Benutzer einen wesentlichen Teil der Leistungsfähigkeit der genutzten

Infrastruktur verfügbar zu machen. Besonders deutlich wird hier aber auch, daß es dazu immer noch optimaler Bedingungen bedarf: die hohe Auslastung bewirkt gerade, daß in diesem empfindlichen Netzwerkbereich die Leistung drastisch einbricht, wenn nicht alles genau paßt. Eine Verbesserung dieser Situation ließe sich zum Beispiel dadurch erreichen, daß dem Benutzer eine CPU exklusiv zur Verfügung gestellt wird. Dies ist jedoch bei vier Prozessoren eine schwierige Angelegenheit; es ist auch fraglich, wieviele Benutzer bereit wären, den entsprechenden Preis für die (zum Teil nicht verbrauchte) CPU-Zeit zu zahlen. Mit der zunehmenden Verbreitung von Rechnern mit mehr Prozessoren (zum Beispiel die NEC SX4 des Rechenzentrums mit 32 Vektorprozessoren) könnte es jedoch durchaus eine sinnvolle Ergänzung zum traditionellen Batch-Betrieb darstellen.

5.2.3 Messungen zwischen einer Workstation und einer Cray

Nach den Messungen zwischen jeweils zwei Rechnern gleicher Größenordnung sollen hier die Resultate von Messungen für den Datenaustausch zwischen einem Supercomputer und einer Workstation untersucht werden. Dieser Fall dürfte auch die höchste Praxisrelevanz haben, da die meisten Benutzer von einer Workstation aus auf die Supercomputer zugreifen. Bei den Meßpunkten (siehe Abbildung 5-11) handelt es sich um eine Kombination aus den Meßpunkten der jeweils paarweisen Messungen der Workstations und Supercomputer. Besondere Bedeutung kommt bei diesen Messungen der Datenkonversion zu. Die Cray Vektorsupercomputer bedienen sich eines eigenen Formats für die Repräsentation von Fließkommazahlen, das vom Standard IEEE-Format abweicht. Dies bedeutet, daß jede Fließkommazahl, die von einer solchen Cray zu einer Workstation geschickt wird, konvertiert werden muß. Dieser Konvertierungsvorgang wird explizit zwischen den Meßpunkten 5 und 6 gemessen. Der konvertierte Datensatz hat eine Größe von 3,217,720 Byte (der Cray-Datensatz ist doppelt so groß, wird aber von 8 Byte Cray-Floats auf 4-Byte IEEE Floats konvertiert).

In diesem Abschnitt werden Messungen über die gesamte Bandbreite der zur Verfügung stehenden Netzwerke präsentiert. Zum Teil wurden diese Messungen zwischen einer Indigo und der C94 durchgeführt, ein weiterer Teil zwischen einer Indigo und der Y-MP. Dies begründet sich in dem Interesse in der knappen verfügbaren Zeit einen möglichst guten Überblick über die verschiedenen für Benutzer interessanten Szenarien zu bekommen. Der Aufwand, alle möglichen Szenarien unter allen Bedingungen zu untersuchen, wäre ungleich größer gewesen; die hier präsentierten Untersuchungen ergeben jedoch schon einen sehr umfassenden Eindruck von den relevanten Einflußgrößen.

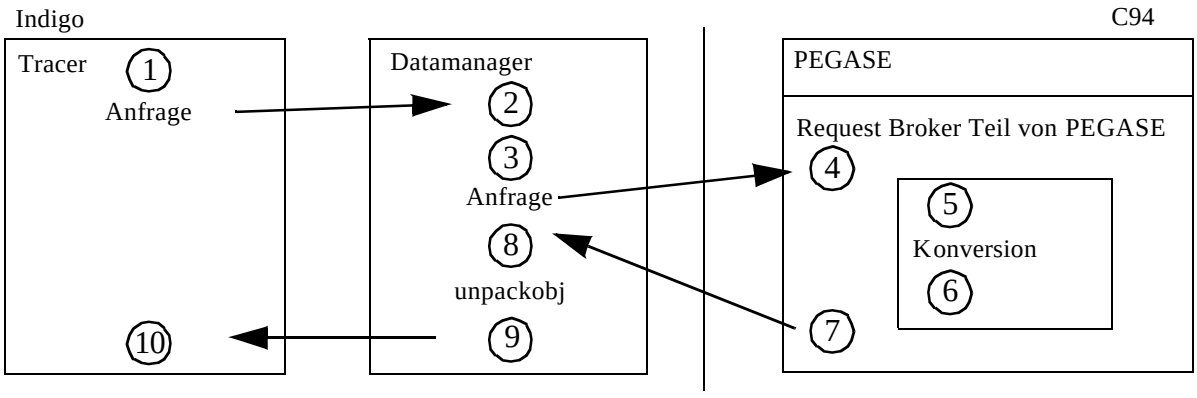

Abbildung 5-11: Meßpunkte der Konfiguration Indigo/C94

Ethernet Messungen

In Tabelle 5-6 sind die Messungen bei einem Datenzugriff über das Ethernet zu sehen. Obwohl das Ethernet nicht das geeignetste Netzwerk ist um auf einen Supercomputer zuzugreifen, ist es doch in der Praxis für die meisten Benutzer die einzige Möglichkeit. Im Hinblick auf die zunehmende Verbreitung schneller Netzwerke auch für den nicht-lokalen Zugriff, werden Techniken mit dieser Geschwindigkeit für einen immer größer werdenden Benutzerkreis interessant. Allerdings zeigen die Ergebnisse sehr deutlich das große Ungleichgewicht zwischen Netzwerkleistung und Leistung der C94. Der im Vergleich lange Aufenthalt im PEGASE-Request Broker des ersten Durchlaufs ist wieder auf die Last der C94 zurückzuführen. Der Unterschied im Netzwerkdurchsatz ist ebenfalls auf das Scheduling-Verhalten

Tracer	lokaler RB	warten im lokalen RB	RB in PEGASE	Konversion	Netzwerk	Netto Durchsatz	Brutto Durchsatz
10 - 1	9 - 2	8 - 3	7 - 4	6 - 5	(8 - 3) - (7 - 4)	Größe / Netzwerk	Größe / Tracer
13.888	13.878	13.459	1.814	0.019	11.645	276 KB/s	231 KB/s
7.901	7.888	7.656	0.034	0.018	7.622	422 KB/s	407 KB/s
14.668	14.652	14.426	0.029	0.020	14.397	223 KB/s	219 KB/s

Tabelle 5-6: C94 zu Indigo, Ethernet, kleiner Datensatz

der C94 zurückzuführen sein. Die etwa 400 KB/s werden jedoch auch von anderen Anwendungen nicht deutlich überschritten. Der in diesem Übertragungsweg liegende stark belastete CISCO-Router ist dafür verantwortlich.

FDDI-Messungen

Bei den FDDI-Messungen geht die Verbindung direkt vom FDDI-Interface der Workstation zum FDDI-Interface der C94. Einziges Zwischenglied ist der bereits früher erwähnte Gigaswitch, der den logischen FDDI-Ring in zwei physikalische Ringe teilt, jedoch nur minimalen Einfuß auf den erreichbaren Durchsatz hat. Die Verteilung der Meßpunkte ist wie im Fall der Ethernet-Messungen. Auch bei diesen Messungen (siehe Tabelle 5-7) wirkt sich die Last der C94 aus: in der Hälfte der Fälle dauert der Request Broker Anteil des PEGASE Prozesses mehr als drei Sekunden. In der anderen Hälfte zeigt sich jedoch eine gleichbleibend gute Performance, die sich auch im Bruttodurchsatz niederschlägt: mit fast 5 MB/s wird das angeforderte Datenobjekt dem Tracer zugänglich gemacht. Dies schließt auch in diesem Fall die Konvertierung der Daten ein, die mit etwa drei Prozent zu Buche schlägt, also bei der Gesamtperformance keine Rolle spielt. Die nötige Konvertierung von Daten ist also kein Grund, einen Prozeß nicht auf einen Cray Vektorsupercomputer auszulagern, wenn der Algorithmus davon profitiert.

Tracer	lokaler RB	warten im lokalen RB	RB in PEGASE	Konversion	Netzwerk	Netto Durchsatz	Brutto Durchsatz
0.780	0.767	0.512	0.028	0.018	0.484	6648 KB/s	4125 KB/s
4.725	4.709	4.550	3.802	0.018	0.748	4301 KB/s	680 KB/s
3.759	3.744	3.593	3.114	3.107	0.479	6717 KB/s	856 KB/s
0.683	0.670	0.456	0.029	0.018	0.427	7535 KB/s	4711 KB/s
0.665	0.658	0.501	0.030	0.019	0.471	6831 KB/s	4838 KB/s
0.664	0.650	0.491	0.040	0.029	0.451	7134 KB/s	4845 KB/s

Tabelle 5-7: C94 zu Indigo, FDDI, kleiner Datensatz

Tracer	lokaler RB	warten im lokalen RB	RB in PEGASE	Konversion	Netzwerk	Netto Durchsatz	Brutto Durchsatz
4.321	4.306	4.143	3.558	3.552	0.585	5500 KB/s	744 KB/s
4.164	4.150	3.840	3.220	3.213	0.620	5189 KB/s	772 KB/s

Tabelle 5-7: C94 zu Indigo, FDDI, kleiner Datensatz

Im Unterschied zu den Ethernet-Messungen wird in den langsameren Fällen die Zeit fast ausschließlich im Request Broker auf der C94 verbracht und nicht beim Verschicken der Daten (Spalte Netzwerk). Dies dürfte damit zusammenhängen, daß die Zeit, die zum Verschicken der Daten gebraucht wird (zwischen 400 und 800 ms) unterhalb der Schedulinggrenze der C94 liegt, d.h. der Vorgang des Sendens wird in der Regel nicht unterbrochen. Bei den Ethernet-Messungen dauerte dieser Vorgang immer mehrere Sekunden, es ist also vorprogrammiert, daß ein solches Senden vom Scheduling unterbrochen und dadurch weiter verlängert wird. Die Nutzung schnellerer Netzwerke macht also den Datenaustausch nicht nur durch die erhöhte Geschwindigkeit schneller, sondern reduziert zusätzlich noch die Anfälligkeit gegenüber anderen Systemvorgängen wie z.B. dem Scheduling.

Um diese Einflüsse weiter zu untersuchen, wurden die gleichen Messungen noch einmal mit einem deutlich größeren Datensatz durchgeführt (23,501,008 Bytes). Aufgrund einer etwas besseren Last-Situation der C94 waren die Resultate dieser Messungen gleichmäßiger als mit dem kleineren Datensatz (siehe Tabelle 5-8). Nur eine der Messungen wurde im Netzwerk gebremst, drei der Messungen wurden vom Scheduling im PEGASE-Request Broker unterbrochen, eine davon bei der Konversion. Sowohl der Nettodurchsatz mit maximal über 8 MB/s als auch der Bruttodurchsatz mit fast 6 MB/s liegen spürbar höher als bei den Messungen mit dem kleinen Datensatz.

Tracer	lokaler RB	warten im lokalen RB	RB in PEGASE	Konversion	Netzwerk	Netto Durchsatz	Brutto Durchsatz
5.417	5.406	3.818	0.177	0.163	3.641	6454 KB/s	4338 KB/s
4.247	4.239	3.065	0.151	0.136	2.914	8064 KB/s	5533 KB/s
14.276	14.257	10.560	0.159	0.145	10.401	2259 KB/s	1646 KB/s
7.594	7.582	6.222	3.173	3.157	3.065	7667 KB/s	3094 KB/s
4.346	4.329	3.144	0.148	0.134	2.996	7844 KB/s	5407 KB/s
5.950	5.936	4.940	2.095	0.139	2.845	8260 KB/s	3949 KB/s
4.067	4.054	3.054	0.155	0.141	2.899	8106 KB/s	5778 KB/s
5.324	5.310	4.310	1.469	0.252	2.841	8272 KB/s	4414 KB/s

Tabelle 5-8: C94 zu Indigo, FDDI, großer Datensatz

Um den Einfluß der Last auf die Messungen möglichst genau zu untersuchen ,wurden die Messungen mit den gleichen Daten noch einmal während der Wartungszeit der C94, also auf einer nahezu leeren Maschine, wiederholt. Die Ergebnisse in Tabelle 5-9 zeigen, daß in der Maximalleistung nur noch geringe Zuwächse zu erreichen sind. Allerdings wird die Maximalleistung öfter erreicht als auf der ausgelasteten C94. Aus Benutzersicht bedeutet dies, daß die Maximalleistung auch auf einer ausgelasteten C94 erreicht werden kann, es aber nur relativ selten gelingen wird. Die schlechten Meßergebnisse im dritten Durchlauf sind durch ein (reproduzierbares) Swappen auf der Workstation verursacht worden.

Genauere Untersuchungen des reinen FDDI-Transfers zwischen einer SGI Indigo und der C94 ergaben im übrigen, daß der im Vergleich zu einer Indigo-Indigo Verbindung niedrigere Nettodurchsatz auf die C94 zurückzuführen ist: sie ist einfach nicht in der Lage mehr Pakete zu verschicken, wohingegen die Indigo mehr Pakete empfangen könnte.

Tracer	lokaler RB	warten im lokalen RB	RB in PEGASE	Konversion	Netzwerk	Netto Durchsatz	Brutto Durchsatz
4.836	4.827	3.181	0.140	0.128	3.041	7728 KB/s	4859 KB/s
7.603	7.223	3.959	0.480	0.128	3.479	6755 KB/s	3091 KB/s
21.401	21.385	10.379	0.141	0.129	10.238	2295 KB/s	1098 KB/s
4.428	4.418	3.154	0.142	0.130	3.012	7802 KB/s	5307 KB/s
4.215	4.202	3.043	0.141	0.129	2.902	8098 KB/s	5575 KB/s
3.950	3.936	2.958	0.141	0.129	2.817	8342 KB/s	5949 KB/s
3.995	3.985	3.005	0.141	0.129	2.864	8205 KB/s	5882 KB/s
3.898	3.886	2.913	0.140	0.128	2.773	8474 KB/s	6028 KB/s

Tabelle 5-9: C94 zu Indigo, FDDI, großer Datensatz, leere C94

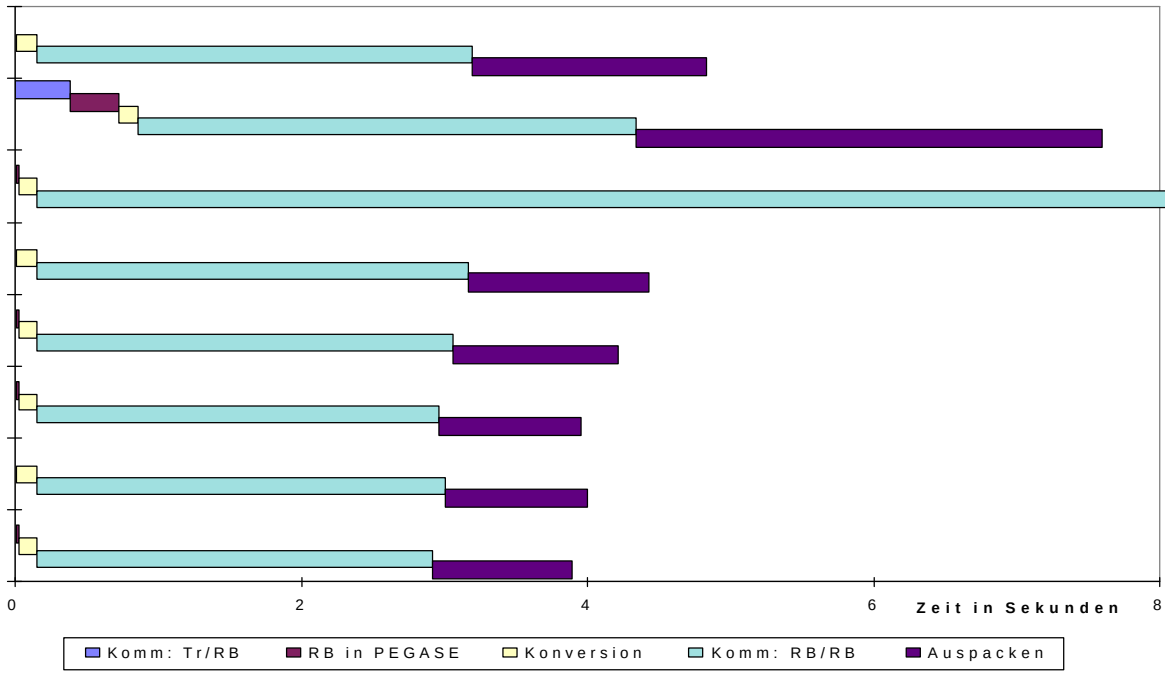

Abbildung 5-12: Graphik der Meßergebnisse aus Tabelle 5-9

In Abbildung 5-12 ist eine graphische Darstellung zu sehen, in der die Zeit dargestellt ist, die in den einzelnen Abschnitten verbracht wurde. Die Skalierung ist so gewählt, daß die wesentlichen Details zu erkennen sind, der Ausreißer in den Meßdaten aus dem dritten Durchlauf geht weit über den rechten Rand hinaus. In dieser Darstellung ist sehr deutlich zu sehen, welch großen Anteil an der Gesamtzeit das Auspacken der Daten auf der Workstation hat: etwa ein Viertel der Zeit wird für diesen Teil aufgewendet. Der größte Teil der restlichen Zeit geht in den tatsächlichen Datentransfer über das Netzwerk. Die Kommunikation zwischen dem Tracer Modul und dem lokalen Request Broker kann vernachlässigt wer-

den, die Zeit, die auf der C94 zum Konvertieren und Einpacken gebraucht wird, ist auch relativ klein im Vergleich zum Auspacken auf der Workstation. Die Rate beim Konvertieren und Einpacken liegt immerhin noch bei über 180 MB/s; zum Vergleich: nur Einpacken auf der C94 450 MB/s, auf der Indigo 20 MB/s. Auch hier zeigt sich deutlich die Überlegenheit der auf Höchstleistung ausgelegten Supercomputer-Architektur.

HIPPI-Messungen

Als letzte Gruppe von Messungen sollen hier noch die für die vom HIPPI-Interface der Y-MP zum FDDI-Interface der Indigo gehende Verbindung dargestellt werden. Diese Verbindung führt vom HIPPI-Interface über einen NSC DX-HI Router in den FDDI-Ring und dann direkt (wiederum über den Gigaswitch) zum FDDI-Interface der Workstation. Die Meßpunkte sind wieder analog zu den anderen Messungen dieses Kapitels, allerdings ist der erste Modul nicht mehr ein Numerikprozeß wie auf dem Computeserver C94, sondern ein Prozeß, der aus einer Datei Daten liest (die Y-MP ist schließlich ein Fileserver). Die zu übertragenden Daten haben eine Größe von 3,182,608 Bytes und werden vor dem Versenden auch vom Cray Fließkomma-Format in das IEEE-Format konvertiert.

Nach den Ergebnissen der Messungen zwischen dem FDDI-Interface der C94 und dem der Indigo sollten eigentlich ähnliche Durchsatzraten erreicht werden können. Allerdings dürfte sich auch wieder die knappe Ausstattung der Y-MP mit Hauptspeicher bemerkbar machen. In Tabelle 5-10 ist denn auch zu erkennen, daß trotz des recht kleinen Datensatzes nur in drei Fällen eine nahezu ungebremste Übertragung von der Y-MP zur Workstation erfolgen konnte. Wenn die Konversion stets unbeeinflußt vom Scheduling blieb, so war doch die Zeit, die der Prozeß im Request Broker auf der Y-MP verbrachte, in den meisten Fällen sehr lang (zwischen 5 und 12 Sekunden). Dabei hätte die eigentliche Arbeit in weniger als 100 ms erledigt werden können.

Die im Netzwerk verbrachte Zeit war jedoch meist etwa gleich lang; dies dürfte entweder darauf zurückzuführen sein, daß die relativ kleine Datenmenge mit wenigen send-Befehlen verschickt wurde, so daß keine Unterbrechung durch das Scheduling auftritt oder daß diese Unterbrechungen relativ gleichmäßig auftreten. In Abbildung 5-13 sind die Ergebnisse der drei schnellsten Messungen zu sehen. Auch bei dieser theoretisch vergleichsweise schnellen Verbindung geht die meiste Zeit im Netzwerk verloren, während etwa ein Viertel mit Ein- und Auspacken sowie Konversion der Daten verbracht werden.

Tracer	lokaler RB	warten im lokalen RB	RB in Read	Konversion	Netzwerk	Netto Durchsatz	Brutto Durchsatz
6.846	6.838	6.626	5.922	0.051	0.704	4520 KB/s	464 KB/s
9.425	9.418	9.280	8.520	0.050	0.760	4187 KB/s	337 KB/s
10.367	10.361	10.223	9.420	0.100	0.803	3963 KB/s	306 KB/s
0.852	0.846	0.710	0.085	0.057	0.625	5092 KB/s	3735 KB/s
13.595	13.576	13.436	12.762	0.052	0.674	4721 KB/s	234 KB/s
1.005	0.989	0.769	0.070	0.056	0.699	4553 KB/s	3166 KB/s
8.828	8.821	8.687	7.804	0.050	0.883	3604 KB/s	360 KB/s
0.815	0.809	0.673	0.066	0.051	0.607	5243 KB/s	3905 KB/s

Tabelle 5-10: Y-MP zu Indigo, HIPPI nach FDDI

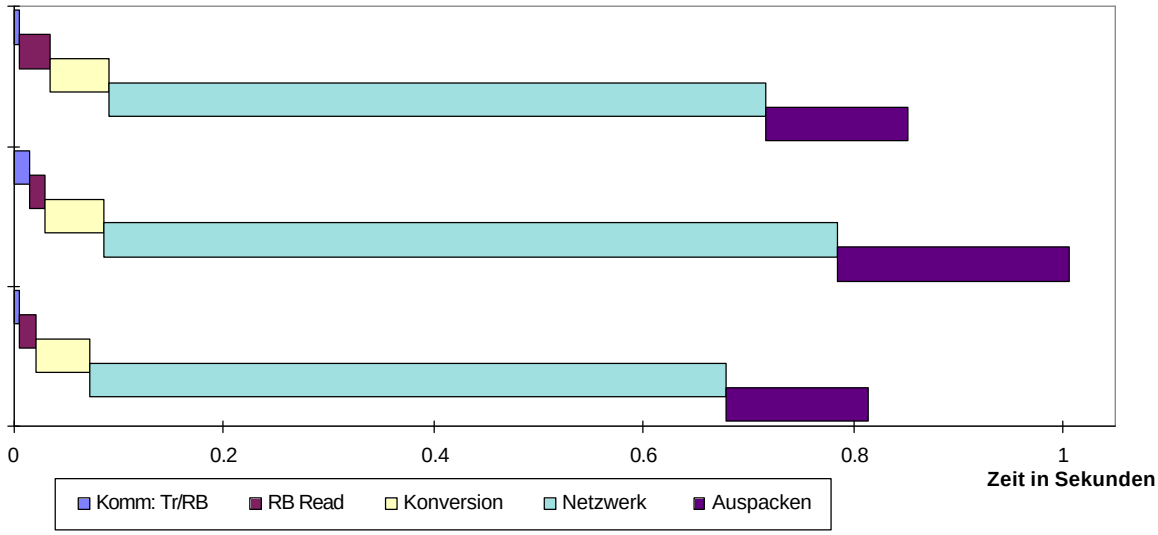

Abbildung 5-13: Graphik der Meßergebnisse aus Tabelle 5-10

Insgesamt bleiben jedoch sowohl der Netto- als auch der Bruttodurchsatz deutlich hinter den Erwartungen zurück. Während zwischen dem FDDI-Interface der C94 und der Workstation bis zu 7,5 MB/s bei einem vergleichbaren Datensatz fließen, sind es zwischen dem HIPPI-Interface der Y-MP und der Workstation maximal knapp über 5 MB/s. Beim Bruttodurchsatz verhält es sich ähnlich: statt bis zu 5,7 MB/s sind es hier nur maximal 3,9 MB/s. Da jedoch sowohl das HIPPI-Interface der Y-MP als auch das FDDI-Interface der Workstation zu einer wesentlich höheren Datenrate in der Lage sind, ist der Grund für die geringere Leistung beim NSC DXE-HI Router von HIPPI auf FDDI zu suchen.

Tracer	lokaler RB	warten im lokalen RB	RB in Read	Konversion	Netzwerk	Netto Durchsatz	Brutto Durchsatz
0.892	0.860	0.615	0.059	0.058	0.556	5724 KB/s	3567 KB/s
0.800	0.793	0.653	0.058	0.057	0.595	5348 KB/s	3978 KB/s
0.730	0.722	0.569	0.071	0.071	0.498	6390 KB/s	4359 KB/s
24.395	24.385	24.247	23.744	23.744	0.503	6327 KB/s	130 KB/s
8.172	8.165	8.028	7.259	7.258	0.769	4138 KB/s	389 KB/s
13.552	13.544	13.403	12.810	12.809	0.593	5366 KB/s	234 KB/s
0.702	0.694	0.557	0.060	0.059	0.497	6403 KB/s	4533 KB/s
0.808	0.801	0.633	0.058	0.058	0.575	5534 KB/s	3938 KB/s

Tabelle 5-11: Y-MP zu Indigo, HIPPI nach FDDI, leere Y-MP

Um auch den letzten Einfluß der Auslastung der Y-MP auf die Meßergebnisse auszuschließen, wurden diese Messungen noch einmal während der Wartungszeit, also auf leerer Maschine (ohne NFS-Daemons) wiederholt. Die Ergebnisse in Tabelle 5-11 bestätigten jedoch im wesentlichen die Ergebnisse der drei schnellen Messungen auf der normal belasteten Y-MP. Immerhin steigt der Nettodurchsatz im Schnitt um etwa 1 MB/s, das heißt auf der ausgelasteten Y-MP wird selbst das Senden der relative kleinen Datenmenge noch vom Scheduling beeinflußt. Auffallend ist jedoch, daß beim vierten bis sechsten Durchlauf trotz leerer Maschine bei der Konversion drastische Verzögerungen auftreten.

5.2.4 Fazit der Messungen

Diese Messungen müssen unter der Randbedingung beurteilt werden, daß sie unter nicht exakt reproduzierbaren Bedingungen stattgefunden haben. Auch lassen sich nicht alle Phänomene und Besonderheiten eindeutig erklären. Allerdings besteht der Sinn dieser Messungen auch nicht darin, absolute Performancewerte zu ermitteln, die dann mit anderweitig gewonnenen verglichen werden sollen. Vielmehr sollen diese Messungen beleuchten, ob das Ziel, ein Kommunikationswerkzeug zu schaffen, das in der Lage ist, High-Performance-Komponenten effizient zu nutzen, verwirklicht worden ist. Das bedeutet, daß unter den Meßergebnissen insbesondere die Relation zwischen der Zeit, die durch das Netzwerk verbraucht wird und der Zeit, die für die Vor- und Nachbereitung des Datentransfers gebraucht wird, relevant ist. Natürlich darf auch das Verhältnis der Messungen unter optimalen Umständen zu denen, die unter den Standardbedingungen stattgefunden haben, nicht vernachlässigt werden. Insbesondere die HIPPI-Messungen sind ein deutliches Zeichen, daß es unerläßlich ist, sich mit den Parametern ausführlich zu beschäftigen, um die teure Technik auch sinnvoll ausnutzen zu können.

Der hohe Grad der Ausnutzung der theoretisch möglichen Bandbreite bei den FDDI Workstation-Workstation und bei den HIPPI Supercomputer-Supercomputer Messungen sind ein deutliches Indiz dafür, daß sich der bei der Optimierung betriebene Aufwand gelohnt hat. Gerade der hohe Anteil des Nettodurchsatzes der zum Beispiel bei der HIPPI-Verbindung auch beim Bruttodurchsatz (also dem für den Benutzer einzig relevanten Durchsatz) übrigbleibt, belegt die hohe Qualität der Datenkommunikation. Vergleicht man diese Raten mit denen der heute meist verwendeten Mechanismen (z.B. ftp oder NFS), so wird deutlich, daß mit dieser interaktiven verteilten Arbeitsweise eine ganz neue Qualität in der Nutzung Einzug hält. Und zwar in einer Weise, die dem Benutzer absolut transparent erscheint und sich einfach nutzen läßt.

Nachdenklich stimmt jedoch der große Einfluß, den die Auslastung der beiden Supercomputer auf die Gesamtperformance hat. In den meisten Fällen sind die Hälfte der Messungen oder mehr deutlich hinter dem zurückgeblieben, was unter optimalen Umständen erreichbar ist. Gerade der Vergleich dieser Werte mit dem Durchsatz, den zwei miteinander über FDDI verbundene Workstations erreichen, läßt Zweifel an der Verträglichkeit des interaktiven Arbeitens mit den derzeit dem Betrieb der Supercomputer zugrundeliegenden Bedingungen: diese werden immer noch überwiegend im Batch-Betrieb gefahren, d.h. ein Benutzer stellt einen Numerik-Auftrag in eine Queue ein, die dann zu gegebener Zeit abgearbeitet wird. Durch das Vorhandensein mehrerer Queues werden die Bedürfnisse verschiedener Benutzer abgedeckt (kleine kurz laufende oder große lang laufende Simulationen) und eine CPU-mäßig optimale Auslastung der Supercomputer sichergestellt.

Allerdings sollte man durchaus darüber nachdenken, ob nicht eine Phase, in der die Benutzer interaktiv eine Reihe von Vorberechnungen (durchaus auch mit größeren Problemen) durchführen, effizienter ist als eine Menge vergleichsweise unkontrollierter Batch-Berechnungen. Die interaktive Arbeitsweise ermöglicht nämlich in vielen Fällen frühzeitig zu erkennen, ob eine Simulation zu den gewünschten Ergebnissen führt oder nicht. In diesen Fällen könnte der Benutzer dann sofort reagieren und eine neue Ausführung mit besseren Parametern starten. Dies spart nicht nur Rechenzeit ein, sondern erlaubt dem Benutzer auch eine deutlich effizientere Optimierung seiner Berechnungen. Insbesondere bei der Entwicklung von Produkten in der Industrie, die ja in der Regel unter einem von den Mitbewerbern erzeugten Zeitdruck erfolgt, kann der Zeitvorteil wesentlich zu kürzeren Produktzyklen und damit größerer Wettbewerbsfähigkeit führen.

Eine Möglichkeit, die in diesen Messungen aufgetretenen lastbedingten Verzögerungen zu reduzieren, bestünde zum Beispiel darin, für bestimmte Zeiten einen Prozeß auf einer CPU fest zu verankern.

Versuche mit erhöhter Priorität auf den beiden Crays (der maximal vom System her an nicht Systemprozesse vergebbaren Priorität) haben nicht zu dem gewünschten Erfolg geführt. Wenn es sich beispielsweise erreichen ließe, daß ein Request Broker Prozeß vom Eintreffen einer Botschaft über das Einpacken bis hin zum Verschicken nicht unterbrochen würde, ließe sich die Effizienz der Datenkommunikation sehr nahe an das Optimum bringen, ohne jedoch die Gesamtperformance für andere Benutzer allzu sehr einzuschränken (diese sicher vorhandene Einschränkung ließe sich über passende Accounting-Mechanismen relativ einfach abgelten).

Die Probleme, die durch die knappe Ausstattung der Y-MP mit Hauptspeicher auftraten, lassen sich allerdings kaum durch solche Mechanismen lösen. So gut dieser Rechner für die reinen Fileserver-Aktivitäten ausgestattet ist, für die Einbeziehung dieser Fähigkeiten in eine interaktive Visualisierungsumgebung scheint er doch kaum geeignet. Dies ist sehr bedauerlich, da gerade die interaktive Einbindung eines Fileservers einem Benutzer den Zugriff auf die Ergebnisse früherer Berechnungen deutlich vereinfachen würde. Für einen dem interaktiven Datenzugriff besser gerüsteten Fileserver wäre eine adäquate Hauptspeicherausstattung unbedingt erforderlich.

Nicht verschwiegen werden soll in diesem Zusammenhang auch, mit welch großem Aufwand diese Meßergebnisse erreicht wurden. Dazu zählen nicht nur die speziellen Implementierungen in der Software, sondern auch die zum Teil äußerst aufwendige Suche nach den optimalen Konfigurationen. Nur durch den engen und guten Kontakt zu den jeweils zuständigen Mitarbeitern des Rechenzentrums ließen sich viele Parameter optimal einstellen und der jeweils beste Weg finden (dazu zählt unter anderem auch das Routing bei Vorhandensein mehrerer Netzwerkinterfaces). Für einen Benutzer außerhalb eines Rechenzentrums dürfte sich dieser Optimierungsvorgang noch wesentlich schwieriger gestalten. Dies sollte jedoch nicht als Abschreckung für Benutzer gesehen werden, sondern vielmehr als Aufforderung an die Rechenzentren und die Hersteller der High-Performance-Komponenten, die enorme Leistung, die in ihrer Ausstattung steckt, auch den außenstehenden Benutzern optimal zur Verfügung zu stellen.

Kapitel 6

Bewertung und Ausblick

6.1 Bewertung

In dieser Arbeit werden zwei Gebiete in Verbindung gebracht, die normalerweise nicht sehr viel miteinander zu tun haben: Datenbanken und Visualisierung.

Die Visualisierung beschäftigt sich damit, aus Daten Informationen einem Betrachter in der bestmöglichen Weise zugänglich zu machen. Das wird deutlich in dem bekannten Spruch: ein Bild sagt mehr als tausend Worte. Übertragen auf die Visualisierung der Ergebnisse komplexer numerischer Berechnungen sagt ein Bild mehr als eine Milliarde Zahlen.

Auf der anderen Seite stehen die Datenbanken: sie werden dazu genutzt, eine große Menge von Daten so zu speichern, daß ein Benutzer möglichst einfach auf sie zugreifen kann. Es gibt Möglichkeiten, in diesen Daten zu suchen und eine ganze Reihe von Operationen auf ihnen durchzuführen (verknüpfen, sortieren usw.), d.h. sie zu analysieren.

Betrachtet man diese beiden Absätze genau, so stellt man jedoch sehr schnell eine grundlegende Gemeinsamkeit dieser beiden Gebiete fest: beide helfen dem Benutzer dabei, die Daten zu verstehen und zu nutzen. Bei der Visualisierung überwiegt dabei der Aspekt der Einsicht, bei den Datenbanken der der Strukturierung und Ordnung. Es wird aber unmittelbar deutlich, daß sich die beiden Gebiete sehr gut ergänzen: wer viele komplexe Daten zu analysieren hat, braucht unbedingt eine Möglichkeit, Ordnung in seinen Daten zu halten. Wer Ordnung in einer großen Menge von Daten zu halten hat, kann durchaus ein Instrument gebrauchen, daß es ihm ermöglicht, sich diese Daten genauer anzuschauen.

Es gibt jedoch noch weitere Unterschiede bei den Begriffen Ordnung und Anschauen in den beiden Gebieten. Bei Datenbanken stellt man sich meist eine große Menge relativ ähnlicher Datensätze oder -objekte vor, die für sich genommen vergleichsweise klein sind (wie zum Beispiel die Personaldaten einer großen Firma). Bei der Visualisierung komplexer Daten handelt es sich meist um sehr große Datensätze, die zwar eine innere Struktur haben (wie z.B. die Elemente bei der Finite-Element-Methode), die aber im Hinblick auf das Gesamtproblem nur eine untergeordnete Rolle spielen. Entsprechend interessiert man sich bei einer Datenbank meist für einen einzelnen Datensatz oder eine Menge von Datensätzen, die ein bestimmtes Merkmal gemeinsam haben. Bei einer Visualisierung wird jedoch in der Regel die Information, die in einem Datensatz steckt, in vergleichsweise aufwendiger Form aufgearbeitet, bevor sie dargestellt wird [11] [12] [13] [20] [56].

Da es sich jedoch sowohl bei der Visualisierung als auch bei den Datenbanken um Felder handelt, die außerordentlich weit reichen, gibt es durchaus sehr starke Berührungspunkte, die es sinnvoll erscheinen lassen, diese beiden Gebiete stärker miteinander in Verbindung zu bringen, als dies bisher der Fall ist. Es gibt mehrere Richtungen, diese beiden Gebiete stärker miteinander zu verbinden:

- zu einer bestehenden Datenbank wird eine Visualisierungsfunktionalität hinzugefügt, die es ermöglicht, die in dieser Datenbank gespeicherten Daten (die oft schon Graphikdaten enthalten: Satellitenbilder oder allgemeine Fotos, Ergebnisse numerischer Berechnungen) direkt darzustellen. Dazu kommen Abfragemöglichkeiten, die versuchen, den Inhalt der Bilder abfragbar zu machen (siehe hierzu [38] [79] und insbesondere [2]).
- die Komplexität einer Datenbank oder die Struktur der in ihr gespeicherten Daten wird graphisch sichtbar gemacht. Dies schließt unter Umständen auch eine graphische (dreidimensionale) Benutzeroberfläche ein, so daß der Benutzer direkt mit den Daten agiert [28].
- einem Visualisierungssystem wird eine Schnittstelle zu einer Datenbank eingebaut. Bei einem modularen System wird in der Regel ein Modul angeboten, der die Möglichkeit bietet, über eine Abfragesprache (z.B. SQL) eine Anfrage an eine Datenbank zu richten und mit den daraus resultierenden Daten zu arbeiten.

Bei dem in dieser Arbeit vorgestellten Ansatz wird jedoch eine vollständige Integration dieser beiden Bereiche angestrebt. Das interne Datenmanagement eines Visualisierungssystems wird mittels einer Datenbank-orientierten Technik durchgeführt. Wie bereits erläutert, handelt es sich hierbei nur um eine relativ einfache Datenbank-ähnliche Datenverwaltung; dennoch entspricht die interne Datenhandhabung der einer normalen Datenbank: damit ein Modul mit einem Datenobjekt arbeiten kann, stellt es eine Anfrage an einen Request Broker, der dann im Gegenzug das gewünschte Objekt (bzw. seine Adresse) zur Verfügung stellt.

Zur Zeit ist diese Anfrage sehr einfach gehalten, da ein Modul nur nach dem Namen eines Objektes fragen kann. Es spricht aber nichts dagegen, an dieser Stelle eine generelle Abfragemöglichkeit einzuführen, die es sowohl Modulen ermöglicht, umfangreichere Abfragen zu stellen, als auch dem Benutzer über die Benutzerschnittstelle. Weiterhin sind in dieser Arbeit bereits die zahlreichen Vorteile einer dedizierten Datenverwaltung im Hinblick auf eine effiziente Datenkommunikation, die Erweiterung in Richtung Persistenz und Cacheing oder Einführung neuer Datentypen erläutert worden.

Der mit einer solchen nicht ganz direkten Datenhandhabung verbundene Zusatzaufwand ist dabei nicht so hoch, daß er die Effizienz dieses Systems in Frage stellen kann. Die Messungen haben gezeigt, daß die Vorteile der Datenverwaltung vielmehr die Gesamteffizienz des Systems erhöhen, da sie dem Benutzer eine Möglichkeit gibt, die Verteilung der Module seiner Visualisierung optimal an die jeweiligen Gegebenheiten anzupassen, ohne daß er sich mit umständlicher Konfigurationsarbeit beschäftigen muß [75] [76].

6.2 Ausblick

Ein großer Teil des in dieser Arbeit aufgezeigten Potentials der Datenbank-orientierten Datenverwaltung ist erst in Ansätzen realisiert. Dazu gehören unter anderem die Persistenz, das Cacheing und die partitionierten Objekte. Diese Bereiche werden intensiv auf ihre Funktionalität hin untersucht werden. Für die industrielle Anwendung müssen die Fragen der Integration einer solchen Umgebung in den Prozeßablauf geklärt werden: wie läßt sich zum Beispiel das Datenmanagement an ein PDM-System anbinden?

Weiterhin sind die Datenobjekte, die bislang verarbeitet werden, immer statisch, selbst wenn es sich um zeitabhängige Daten handelt. Erste Konzepte für die Handhabung von Datenströmen, wie sie zum Beispiel bei Audio- und Videodaten vorkommen, sind schon erarbeitet worden. Dazu erscheint es sinn-

voll, parallel zum Request Broker einen Prozeß einzuführen, der kontinuierlich Daten überträgt und dabei automatisch die in einem Ringpuffer organisierten Datenströme verteilt.

Selbstverständlich wird auch die bestehende Implementierung auf Verbesserungsmöglichkeiten untersucht werden. Der Nachteil der Datenbank-orientierten Datenverwaltung gegenüber den Standardsystemen soll minimiert werden; wesentlich dabei ist, daß die Vorteile in vollem Umfang erhalten bleiben müssen. Die Implementierung ist von Beginn an auf größtmögliche Portabilität ausgelegt worden und läuft bereits auf allen wesentlichen Unix-Plattformen. Eine weitere Portierung auf nicht-Unix-Plattformen wird einige Anpassungen erforderlich machen.

Die bisher durchgeführten Entwicklungen haben ihren Ursprung im Bereich des Supercomputing und der verteilten Visualisierung. Das Konzept ist jedoch nicht auf Visualisierung beschränkt, sondern kann vielmehr als generelles Werkzeug zur Verteilung von Aufgaben auf mehrere Rechner dienen (die Art diese Aufgaben ist im wesentlichen bestimmt durch einen nicht zu hohen Grad der Granularität). Dabei kann aufgrund der COVISE-Gesamtarchitektur immer gleich kollaborativ gearbeitet werden. Bei dieser Ausdehnung des Anwendungsbereiches wird jedoch darauf geachtet werden, nicht mit Standards wie CORBA oder Anwendungen wie AVS/Express in Konkurrenz zu treten, sondern vielmehr mit ihnen zusammenzuarbeiten, wo es sinnvoll ist und sie zu ergänzen, wo die Stärken der hier realisierten Datenbank-orientierten Datenverwaltung (wie z.B. schnelle Verfügbarkeit auch auf exotischen Rechnern wie der Cray T3E oder der NEC SX4 sowie die effiziente Ausnutzung dieser teuren Rechner) zum Tragen kommen.

Auch die Integration mit echten relationalen oder objekt-orientierten Datenbanken ist bislang noch nicht weiter verfolgt worden. Hierbei ergeben sich interessante Kombinationsmöglichkeiten mit lokalen oder ebenfalls verteilten Datenbanken und der Unterstützung von Visualisierungsfunktionalität sowohl für die Visualisierung von Daten in den Datenbanken als auch der Strukturen dieser Datenbanken.

Die Leistungsfähigkeit des hier präsentierten Ansatzes ist sicher noch nicht ausgeschöpft, so daß sich ein großes Feld an weiteren Aufgaben finden läßt, die von diesem Ansatz profitieren können. Wesentlich ist dabei, daß das inhärent Datenverwaltungs-orientierte Konzept es sehr leicht macht, mit Datenbanken und ihrer Funktionalität zusammenzuarbeiten und dabei auch die enge Integration mit der Anwendung unterstützt.

Kapitel 7

Literatur

[1] Abram, G. ; Treinish, L. ; “An Extended Data-Flow Architecture for Data Analysis and Visualization“, Proceedings Visualization ‘95, G. Nielson, D. Silver (Eds.), pp. 263-270, IEEE Computer Society Press, Los Alamitos, 1995

[2] Aiken, A.; Chen, J. ; Lin, M. ; Spalding, M.; Stonebraker, M. ; Woodruff, A.; “The Tioga-2 Database Visualization Environment“, *Database Issues for Data Visualization, Lecture Notes in Computer Science*, Volume 1183, pp. 181 - 207, Wierse, Grinstein, Lang (Eds.), Springer, 1996

[3] Arya, M. ; Swanberg, D ; Vasudevan, V. ; Wierse, A. ; “Database and Visualization: System Integration Issues“, “Database Issues for Data Visualization“, *Lecture Notes in Computer Science*, Volume 871, pp. 16 - 24, Lee, Grinstein (Eds.), Springer, 1994

[4] AVS/Express, Developer‘s Reference, Advanced Visual Systems, Waltham, MA, Aug. 1995

[5] AVS User’s Guide Release 4, Waltham Mass., Advanced Visual Systems Inc. 1992

[6] AVS Technical Overview, Advanced Visual Systems, Waltham, MA, Oct. 1992

[7] Bartz, D. (Ed.) ; „Visualization in Scientific Computing ’98“, Springer-Verlag, Wien, 1998

[8] Bell, D. ; Grimson, J. ; „Distributed Database Systems“, Addison Wesley, Workingham, 1992

[9] Burnett, M. ; Goldberg, A. ; Lewis, T. (Eds.) ; „Visual Object-Oriented Programming“, Manning, Greenwich, 1994

[10] Causse S. ; Juaneda, F. and Grave, M. ; „Partitioned objects sharing for visualization in distributed environments“, in Rosenblum, L et al., *Scientific Visualization, Advances and Challenges*, pp. 251 - 264, Academic Press, London, 1994

[11] Cox, M. ; Ellsworth, D. ; „Application-Controlled Demand Paging for Out-of-Core Visualization“, in Yagel, R. ; Hagen, H. (Eds.) ; *Visualisation 1997, Proceedings*, pp. 235 - 244, IEEE Computer Society, Los Alamitos, 1997

[12] Earnshaw, R.A. ; Wiseman, N. ; „An introductory Guide to Scientific Visualization“, Springer-Verlag, Berlin, 1992

[13] Earnshaw, R.A. ; Watson, D. ; „Animation and Scientific Visualization, Tools & Applications“, Academic Press, London, 1993

[14] Ehnis, K. ; Lang, U. ; Loebich, I. ; Pilz, M. ; Rühle, R. ; „DFN-RSYST, Verteilung einer Modell- und Methodendatenbank mit Datenbasis im offenen Netz“, IKE, Stuttgart 1985

[15] Ensight User Manual for Version 5.2, Computational Engineering International Inc., Research Triangle Park, NC, 1994

[16] Fejes, L. ; Johannsen, G. ; Strätz, G. ; „Hierarchical Data Structure for Dynamic Visual Systems“ in Knuth, E. ; Wegner, L.M. (Eds.) , *Visual Database Systems*, pp. 219 - 234, North-Holland, Amsterdam, 1992

[17] Geiben, M. ; Rumpf, M. ; „Visualization of Finite Elements and Tools for Numerical Analysis“, Report 9, SFB256, Bonn 1991, erschienen in: F.H. Post, A.J. Hin (Eds.), *Advances in Scientific Visualization*, Springer Lecture Series Band, Workshop Proceedings, 1993

[18] Gentzsch, W. ; Harms, U. ; „High-Performance Computing and Networking“, Proceedings, Volume I und II, Springer-Verlag, Berlin, 1994

[19] Globus, A. ; “Perspectives on the IRIS Explorer Visualization Environment“, Computer Sciences Corporation Report RNR-91-021, 1992

[20] Göbel, M. ; Müller, H. ; Urban, B. ; „Visualization in Scientific Computing“, Springer-Verlag, Wien, 1995

[21] Gray, J. ; Reuter, A. ; „Transaction processing: concepts and techniques“, Morgan Kaufmann Publishers, San Mateo, California, 1993

[22] Haas, P. ; Christ, P. ; “Networking Issues in PAGEIN: The “N“ of “HPCN“ “, *Lecture Notes in Computer Science*, W. Gentzsch, U. Harms, Vol. 797, pp. 86 - 93, Springer Berlin, 1994

[23] Hege, H.-C. ; Polthier, K. ; „Visualization and Mathematics“, Springer Verlag, Berlin, 1997

[24] Hocke, M. ; Otter, M. ; „Entwicklung integrierter Programmsysteme mit RSYST“, Rechenzentrum der Universität Stuttgart, RUS-4, März 1990

[25] Hocke, M. ; Seybold, J. ; Wagner, U. ; „Visualisierung und Animation von Mehrkörpersystemen unter Verwendung eines objektorientierten Geometrie-Datenmodells“, *Forschungs- und Entwicklungsberichte RUS-18*, Rechenzentrum der Universität Stuttgart, 1993

[26] IRIS Explorer User’s Guide, Release 3, Silicon Graphics, 1995

[27] IRIS Explorer 2.0, Technical Report, Silicon Graphics Computer Systems, Mountain View 1992

[28] Keim, D.; “Enhancing the Visual Clustering of Query-Dependent Database Visualization Techniques Using Screen-Filling Curves“, in *Database Issues for Data Visualization, Lecture Notes in Computer Science*, Volume 1183, pp. 181 - 207, Wierse, Grinstein, Lang (Eds.), Springer, 1996

[29] Khanna, R. (Ed.) ; “Distributed Computing, Implementation and Management Strategies“, Prentice Hall, Englewood Cliffs, 1994

[30] Krause, E. ; Jürgen, W. (Eds.), „High-Performance Computing in Science and Engineering ’98“, Springer-Verlag, Berlin, 1999

[31] Lang, U. ; „Visualisierung wissenschaftlicher Daten“, Manuskript zur Vorlesung, Rechenzentrum der Universität Stuttgart, 1995

[32] Lang, U. ; Lang, R. ; Rühle, R. ; „Scientific Visualization in a Supercomputer Network at RUS“, *Computer & Graphics* Vol. 17, No. 1, pp. 15-22, Pergamon Press Ltd., 1993

[33] Lang, U. ; Grave, M. ; „Data Structures in Scientific Visualizaion“, in Hagen, H. ; Müller, H. ; Nielson, G.M. ; *Focus on Scientific Visualization*, pp. 85 - 102, Springer-Verlag, Berlin, 1993

[34] Lang, U. ; Lang, R. ; Rühle, R. ; „Visualisation in a Software System for Scientific Computing“, in F.H.Post, A.J.S. Hin (Eds.), *Advances in Scientific Visualization*, Springer-Verlag Berlin, 1992

[35] Lang, U. ; Rühle, R. ; „Visualisierung von Supercomputerberechnungen am netzintegrierten Ingenieurarbeitsplatz“, in Meuer, H. (Hrsg.), *Heterogene Netze und Supercomputer*, pp. 121 - 133, Springer-Verlag Berlin, 1992

[36] Lang, U. ; Rühle, R. ; “Scientific Application Environments“; Positional Paper, SIGGRAPH 90, Workshop on Data Structures and Access Software for Scientific Visualization, August 1990

[37] Lang, U. ; „Modularisierungskonzepte bei Simulationsproblemen in integrierten Programmsystemen“, *Forschungs- und Entwicklungsberichte RUS-2*, Rechenzentrum der Universität Stuttgart, 1989

[38] Lee, J. ; Grinstein, G. (Eds.) ; „Database Issues for Data Visualization“, *Lecture Notes in Computer Science*, Volume 871, Proceedings IEEE Visualization ’93 Workshop, San Jose, Springer-Verlag Berlin, 1994

[39] Löffler, K ; „Ein Datenmodell für wissenschaftlich-technische Anwendungen“, *Forschungs- und Entwicklungsberichte RUS-1*, Rechenzentrum der Universtität Stuttgart, 1988

[40] Lorenson, W.E. and Cline, H.E., „Marching Cubes: A high Resolution 3D Surface Construction Algorithm“, in: Stone, M.C. (Ed.), *Computer Graphics (SIGGRAPH ’80 Proceedings)*, no. 3 in vol. 14, pp. 2-9, Juli 1980

[41] Merazzi, S. ; Bonomi, E. ; „Graphic Representation of Numerical Simulations“ in Thalman, D. (Ed.), Scientific Visualization and Graphics Simulation, pp. 149 - 160, Jon Wiley & Sons, Chichester, 1990

[42] Meyer-Wegener, K. ; „Multimedia-Datenbanken“, B.G. Teubner, Stuttgart, 1991

[43] MP/Express Preliminary Specification, *http://www.avs.com/techpapers/xp/mp-express/white-paper.html*

[44] Mullender, S. ; (Ed.), “Distributed Systems, Second Edition“, Addison Wesley, New York, 1993

[45] Nielson, G.M. ; Shriver B. and Rosenblum, L.J., „Visualization in Scientific Computing“, IEEE Computer Society Press, Los Alamitos, 1990

[46] „Object Management Architecture Guide“, Object Management Group (OMG), Framingham, MA, 1990

[47] Perdue, J. ; „High Bandwidth Interactivity and Super Networks“ in Mendez, R.H. (ed.) ; Visualization in Supercomputing, pp. 80 - 99, Springer-Verlag, New York, 1990

[48] Rabenseifner, R. et al. , „Das DFN Remote Procedure Call Tool, Benutzerhandbuch, Band 1 , Release 1.0.60 beta”, Rechenzentrum der Universität Stuttgart, 1994

[49] Rantzau D. ; Thomas P., “Parallel CFD-Simulations in a Collaborative Software Environment Using European ATM-Networks”, in: *Proceedings of the Parallel Computational Fluiddynamics `96*, Capri, 1996

[50] Rantzau D. ; Lang U., “A Scalable Virtual Environment for Large Scale Scientific Data Analysis”, in: *Future Generation Computer Systems 14 (1998)*, pp. 215-222, Elsevier Science, 1998

[51] Resch M. ; Rantzau D. ; Berger H. ; Bidmon K. ; Keller R. ; Gabriel E., “A Metacomputing Environment for Computational Fluid Dynamics”, in: *Proceedings of the 10th Parallel Computational Fluid Dynamics Conference (Parallel CFD '98)*, Hsinchu, Taiwan 11.-14. May 1998, to be published

[52] Rill, S. ; Grosso, R. ; “Future Aerospace Working Scenarios Using High Speed Networks and Supercomputers Applied to Flow Simulation for Complete Aircraft”, *Lecture Notes in Computer Science*, W. Gentzsch, U. Harms, Vol. 797, pp. 60 - 69, Springer Berlin, 1994

[53] Rühle, R. ; „RSYST - Ein Softwaresystem zur Integration von Daten und Programmen zur Simulation wissenschaftlich-technischer Systeme“, Rechenzentrum der Universität Stuttgart, RUS-5, März 1990

[54] Rumbaugh, J. ; Blaha, M. ; Premerlani, W. ; Eddy, F. ; Lorensen, W. ; „Object-Oriented Modeling and Design“, Prentice-Hall, Englewood Cliffs, 1991

[55] Rumpf, M. ; Schmidt, A. et al. ; „GRAPE, GRaphics Application and Programming Environ-

ment“, SFB256, No. 8, Bonn, 1989

[56] Scateni, R. ; van Wijk, J. ; Zanarini, P. (Eds.) ; „Visualization in Scientific Computing ’95”, Springer-Verlag, Wien 1995

[57] Schlecht, B. ; „Entwurf und Implementierung eines interaktiven anwendungsunabhängigen Grafiksystems“, *Forschungs- und Entwicklungsberichte RUS-3*, Rechenzentrum der Universität Stuttgart, 1989

[58] Schmidt, D. ; „Persistente Objekte und objektorientierte Datenbanken“, Carl Hanser Verlag, München/Wien, 1991

[59] Schroeder, W. ; Lorensen, W. ; Montanaro, G. ; Volpe, C. ; “VISAGE: An Object-Oriented Scientific Visualization System“, *Visualization `92, Proceedings*, Boston, MA

[60] Schroeder, W. ; Martin, K. ; Lorensen, W. ; “The Visualization Toolkit: An Object-Oriented Approach to 3D-Graphics“, Prentice Hall, Upper Saddle River, NJ, 1996

[61] Schroeder, W. ; Martin, K. ; Lorensen, W. ; “The Design and Implementation of An Object-Oriented Toolkit For 3D Graphics And Visualization“, in Yagel, R. ; Nielson, G.M. (Eds.), *Visualization `96, Proceedings*, pp. 93 - 100, IEEE Computer Society Press, Los Alamitos, 1996

[62] Seybold, J. ; „Angewandte objektorientierte Modellierung von Mehrkörpersystemen für die parametrische Visualisierung, dynamische Simulation und Optimierung“, *Forschungs- und Entwicklungsberichte RUS-40*, Rechenzentrum der Universität Stuttgart, 1998

[63] Seybold, J. ; „Spezifikation und Implementierung eines Moduls DFNMDL zur Verteilung von RSYST-Moduln auf verschiedene Plattformen“, *Interne Mitteilung*, Rechenzentrum der Universität Stuttgart, 1996

[64] Seybold, J. ; Rühle, R. ; „Biomechanische Computersimulation und Visualisierung“ in: Arbeitsgemeinschaft Simulation in der Gesellschaft für Informatik(ASIM): *Mitteilungen aus den Arbeitskreisen: Heft Nr. 46*, pp, 105-116, Otto-von-Guericke-Universität Magdeburg, 1995

[65] Sloman, M. ; (Ed.), “Network and Distributed Systems Management“,Addison Wesley, Workingham England, 1994

[66] Spaniol, O. ; „Betriebserfahrungen und Messungen an einem großen FDDI-Netz ...und sich daraus ergebende Konsequenzen“, in Encarnaçao, J. ; *Telekommunikation und multimediale Anwendungen der Informatik*, pp. 22 - 34, Springer-Verlag, Berlin, 1991

[67] Stevens, A. ; “C++ Database Development“, MIS:Press, New York, 1992

[68] Stroustrup, B. ; „The C++ Programming Language“, 2. Auflage, Addison-Wesley, 1991

[69] „The Common Object Request Broker Architecture and Specification“, Object Management Group (OMG)

[70] Treinish, L.A. ; “The role of data management in discipline-independent datavisualization“, „Extracting Meaning from Complex Data: Processing, Display, Interaction“,Edward. J. Farrell Editor, Proc. SPIE 1259, 261-271 (1990)

[71] Upson, C. ; Faulhaber, T. ; Kamins, D. ; Laidlaw, D. ; Schlegel, D. ; Vroom, J. ; Gurdwitz, R. ; van Dam, A. ; „The Application Visualization System: A Computational Environment for Scientific Visualization“, *IEEE Computer Graphics and Applications*, Volume 9, Number 4, pp. 30-42, July 1989

[72] User Guide, AIX/Visualization Data Explorer/6000, Version 3.1, IBM Corporation, Yorktown Heights, 1995

[73] Volle, C ; Seybold, J. ; Rühle, R. ; „RSYST User’s Guide Version 3.6.0“, *Forschung- und Ent*

wicklungsberichte RUS-34, Rechenzentrum der Universität Stuttgart, 1996

[74] Wierse, A. ; Rumpf, M. ; „GRAPE, Eine interaktive Umgebung für Visualisierung und Numerik“, *Informatik, Forschung und Entwicklung*, Springer 7, pp 145 - 151, 1992

[75] Wierse, A. ; Lang, U. ; Rühle, R. ; “Architectures of Distributed Visualization Systems and their Enhancements“, *Workshop Papers of the Fourth Eurographics Workshop on Visualization in Scientific Computing*, Abingdon, UK, April 1993

[76] Wierse, A. ; Lang, U. ; Rühle, R. ; “A System Architecture for Data-oriented Visualization“, “Database Issues for Data Visualization“, *Lecture Notes in Computer Science*, Volume 871, pp. 148 - 159, Lee, Grinstein (Eds.), Springer, 1994

[77] Wierse, A. ; “Performance of the COVISE visualization system under different conditions“, in *Visual Data Exploration and Analysis II*, Georges G. Grinstein, Robert F. Erbacher, Editors, Proc. SPIE 2410, 218-229, 1995

[78] Wierse, A. ; Lang, R. ; Lang, U. ; Nebel, H. ; Rantzau, D. ; “The Performance of a Distributed Visualization system“, *Visualization Methods in High Performance Computing and Flow Simulation*, W. Borchers, G. Domik, D. Kröner, R. Rautmann, D. Saupe, VSP-International Science Publishers, Zeist 1996

[79] Wierse, A. ; Grinstein, G. G. ; Lang, U. (Eds.) ; “Database Issues for Data Visualization“, *Lecture Notes in Computer Science*, Volume 1183, Proceedings IEEE Visualization ’95 Workshop, Atlanta, Springer-Verlag Berlin, 1996

[80] Wössner, U. ; „Implementierung von Visualisierungsmethoden für unstrukturierte Gitter in einer Multi-Block-Umgebung“, Studienarbeit im Fach Angewandte Informatik, Rechenzentrum der Univsersität Stuttgart, 1996

[81] Wössner U. ; Rantzau D. ; Rainer D., „Interactive Simulation Steering in VR and Handling of Large Datasets“, in: *Proceedings of the IEEE Youth Forum in Computer Science and Engineering (YUFORIC) Germany '98*, Stuttgart, 16-18th June 1998